Diogo Pinhal Ribeiro

Licenciado em Ciência e Engenharia Informática

Middleware para controlo cooperativo a partir de uma rede de dispositivos móveis

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Hervé Miguel Cordeiro Paulino, Prof. Auxiliar, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Júri

Presidente: Prof. Paulo Orlando Reis Afonso Lopes, FCT-UNL
Vogais: Prof. Rolando da Silva Martins, FCUP
Prof. Hervé Miguel Cordeiro Paulino, FCT-UNL

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2018

Middleware para controlo cooperativo a partir de uma rede de dispositivos móveis

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Aos meus pais.

Agradecimentos

Agradeço ao meu orientador, Hervé Paulino, pelo apoio e orientação prestada nesta dissertação.

Agradeço também ao departamento de Informática da FCT-UNL e a todos os seus docentes por tudo o que me ensinaram nestes 5 anos.

Um forte agradecimento ao projeto Hyrax (CMUP-ERI/FIA/0048/2013) pelo financiamento desta dissertação.

Por último, um grande agradecimento à minha família pelo apoio que me deram na minha formação académica.

Resumo

Os dispositivos móveis são atualmente uma das principais ferramentas na nossa sociedade, devido às suas grandes capacidades a nível computacional e à sua portabilidade. Para além de comunicação, estes dispositivos são utilizados em variadas atividades do dia-a-dia. A crescente capacidade computacional dos dispositivos móveis tem fornecido a base para o desenvolvimento de novos tipos de aplicações de modo a melhorar a qualidade de vida dos seus utilizadores. Nesta dissertação exploramos mais um desses casos, fornecendo uma forma de suportar a cooperação entre dispositivos.

A solução desenvolvida nesta dissertação oferece um *middleware* para controlo cooperativo. Foi concebida uma *framework* genérica que permite a dispositivos móveis, com o sistema operativo Android, combinarem os seus comandos de forma a controlarem em conjunto uma aplicação. O sistema tem como função receber fluxos de informação fornecidos pelos dispositivos móveis, agrega-los e retornar a combinação das informações enviadas pelos dispositivos. Esse resultado final é depois aplicado numa aplicação destino, independente do sistema.

Para validação foi concebido todo o sistema que permite a cooperação entre dispositivos e foi desenvolvida uma aplicação Android de modo a testar o sistema implementado. Foi utilizado como caso de estudo, ou seja, como aplicação destino, um jogo desenvolvido pela Faculdade de Ciências e Tecnologias, semelhante ao jogo Pong, de modo a obter resultados.

Os resultados obtidos mostram que a solução desenvolvida apresenta o comportamento desejado em ambientes com um número considerável de dispositivos conectados. Comprovou-se que o sistema apresenta tempos de processamento de informação relativamente baixos, o que permite que a frequência de informação enviada para a aplicação destino seja suficientemente alta para que se consiga o comportamento desejado. Foi ainda possível verificar que o consumo energético do protótipo desenvolvido é baixo, o que permite a sua utilização em eventos de curta e média duração.

Palavras-chave: Cooperação, Android, Agregação Distribuída, Controlador Remoto

Abstract

Mobile devices are currently one of the main tools in our society due to their great computational capabilities and portability. In addition to communication, these devices are used in a variety of day-to-day activities. The increasing computational capacity of mobile devices has provided the basis for the development of new types of applications in order to improve the quality of life of its users. In this dissertation we explore one more of these cases, providing a way to support cooperation between devices.

The solution developed in this dissertation offers a middleware for cooperative control. A generic framework has been designed that allows mobile devices, with the Android operating system, to combine their commands to control an application together. The system is intended to receive information flows provided by mobile devices, aggregate them and return the combination of the information sent by the devices. This end result is then applied to a target application.

For validation a whole system that allows the cooperation between devices was designed and an Android application was developed in order to test the implemented system. A game developed by the Faculdade de Ciências e Tecnologias, similar to the Pong game, was used as a case study, that is, as a target application, in order to obtain results.

The obtained results show that the developed solution presents the desired behavior in environments with a considerable number of connected devices. It has been found that the system has relatively low information processing times, which allows the frequency of information sent to the target application to be high enough to achieve the desired behavior. It was also possible to verify that the energy consumption of the developed prototype is low, which allows its use in short and medium duration events.

Keywords: Cooperation, Android, Remote Controller, Distributed Aggregation

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Capítulo 1

Introdução

1.1 Motivação

O número de dispostivos móveis, como *tablets* e *smartphones*, tem vindo a aumentar na sociedade, tornando-se cada vez mais importante no quotidiano. De acordo com a previsão da Cisco [1], estima-se que em 2021 o número destes dispositivos rondará os 12 mil milhões, quando em 2016, o número era apenas 8 mil milhões. Isto significa que é esperado um aumento de 50% num espaço de 5 anos. Com estes valores podemos perceber a importância que estes assumiram na nossa sociedade.

Para além das suas capacidades de comunicação, os dispositivos mais recentes contam com grandes capacidades de computação e elevados níveis de armazenamento. Já existem no mercado dispositivos com processadores octa-core, com 8GB de memória RAM, armazenamento interno de 256GB, módulos de comunicação como Wi-Fi 802.11 [2] e Bluetooth 5.0 [3]. Um exemplo destes é o OnePlus 6 [4]. Alguns dispositivos, chegam a ser superiores a computadores existentes no mercado. Para além das suas capacidades, de computação, contam ainda com outras funcionalidades, como câmaras fotográficas ou GPS.

Com o aumento da capacidade dos seus dispositivos, os utilizadores, tal como os programadores, procuram novas formas de tirar partido dessas novas tecnologias e uma das formas de as usar, é fazer com que os vários dispositivos cooperem entre si de modo a explorar novas formas de interação e já foram feitos projetos à volta deste tema, tais como [5] e [6].

Vajk et al. [7] investigaram a forma de como um grande grupo de pessoas joga utilizando um dispositivo móvel como controlador. Para o controlador usaram-se os dispositivos que comunicam com o computador que suporta o jogo através da tecnologia Bluetooth. Foi utilizado um pequeno jogo de corridas de carros em que cada utilizador

controla um desses veículos. O jogo foi projetado de maneira a que todos os utilizadores pudessem ver toda a pista. A reação dos intervenientes foi de felicidade e prazer ao jogar um jogo tão simples com outras pessoas, explorando assim uma nova experiência social.

J. Leikas et al [8], exploraram a interação entre pessoas quando estas partilham a mesma aplicação. Para isso, utilizou-se um jogo e os seus intervenientes utilizaram os seus dispositivos móveis como controladores. Concluiu-se que este tipo de interações sociais é mais eficaz quando os participantes são amigos/conhecidos.

1.2 Problema

A finalidade desta dissertação é oferecer uma forma de cooperação entre dispositivos móveis, demonstrando ser possível que vários utilizadores, em simultâneo, através dos seus dispositivos, pensam chegar a um objetivo comum. Esta poderá ser usada não só em jogos, para controlar personagens, como também noutros eventos de lazer, por exemplo num concerto, controlando as luzes ou o som, ou no cinema para controlar as câmaras, tornando a experiência dos utilizadores mais interativa e dinâmica. Tendo em conta que nos dias de hoje, a população em geral procura sempre novas formas de diversificar as suas experiências e ao mesmo tempo partilhar essas mesmas com outros, a partir da cooperação é possível obter essa diversificação.

Para se fornecer a funcionalidade pretendida, desenvolveu-se uma *framework* genérica que recebe fluxos de informação enviados pelos dispositivos móveis e agrega esses num único. Estes fluxos são combinados seguindo um algoritmo fornecido pelo utilizador da *framework*. O resultado da agregação é um fluxo de comandos a ser recebido por uma aplicação destino.

Um dos exemplos mais fáceis de se perceber será a cooperação em jogos. Ou seja, num caso comum, os vários utilizadores terão de trabalhar em conjunto de modo a atingir o objetivo final do jogo.

Seguem-se alguns exemplos de aplicação desta dissertação:

Jogo do *Pong* [9] Desenvolvido pela empresa Atari [10], é um jogo em 2D que simula o jogo Ténis de Mesa. Um jogador controla uma barra vertical colocada no lado esquerdo do ecrã, movendo-a verticalmente, ou seja, para cima e para baixo, e o segundo jogador controla uma barra semelhante mas no lado oposto do ecrã. Os jogadores usam as suas barras para acertar numa bola e o objetivo será fazer com que a bola passe para o lado do adversário e que este não a consiga mandar de volta.

No contexto do projeto, um grupo de utilizadores controlará uma das barras verticais, enquanto o segundo grupo controlará a barra oposta. Sendo assim, os utilizadores têm de cooperar entre si de modo a vencer o grupo adversário, para isso utilizarão os seus dispositivos móveis como controladores. Vence o grupo que conseguir cooperar melhor e fazer com que o grupo rival não consiga devolver a bola.

Essa cooperação pode ser feita através da inclinação do dispositivo, ou seja, quando os utilizadores inclinarem o seu controlador para um dos lados, será para essa posição que a barra irá mover-se. Mas surgem outros problemas de como, por exemplo, saber qual a velocidade que a barra se deve mover ou para qual lado a se irá mover, se utilizadores diferentes escolherem direções diferentes.

Para resolver este tipo de problemas terão de ser feitas operações matemáticas com os valores recebidos, e neste caso, uma das operações possíveis será usar a média de valores que são emitidos pelos dispositivos, sendo esses valores enviados através da inclinação do dispositivo e da intensidade desses movimentos. A média de valores seria então o comando final a ser recebido pelo jogo.

Jogo *Arkanoid* [11] O jogo, anteriormente referido, não é o único onde pode ser aplicada esta forma de cooperação, isto é, existem outros jogos no mercado em que é possível que um grupo de pessoas coopere de modo a vencer.

Outro exemplo é o *Arkanoid*, que é um jogo 2D semelhante ao *Pong*, mas em que não há adversário, tem sim o objetivo de eliminar obstáculos. Neste é também possível outro tipo de funcionalidades em que jogos como o *Pong* não exigem, como um botão de disparo.

Em comparação com o *Pong*, poderá ter a mesma forma de mover o jogador, ou seja, inclina-se para um lado ou outro para mover a barra, mas também há a hipótese de clicar num botão de modo a disparar uma bala.

É também possível usar esta plataforma em jogos de corrida, em que o grupo de jogadores controlariam a direção de um veículo.

Outras atividades lúdicas É possível também a cooperação de *smartphones* noutras situações que não jogos, por exemplo, num concerto mover as luzes do palco consoante a vontade do público. Se todos os utilizadores inclinarem os seus dispositivos para um dos lados, as luzes apontariam para esse lado ou se um número mínimo de pessoas levantar o seu dispositivo as luzes acendem ou apagam. Isto irá depender da criatividade tanto do programador como do artista. Contudo neste exemplo a escala será maior, pois num concerto o número de utilizadores será muito maior e isso é um fator a ter em conta.

Outro exemplo de utilização desta tecnologia seria no cinema, os utilizadores em conjunto poderiam controlar a câmara que observa o filme. Neste exemplo, como a escala seria apenas na ordem das dezenas ou centenas.

Esta dissertação tem como desafios principais: a agregação distribuída dos fluxos transmitidos pelos dispositivos móveis, encontrando assim um fluxo final a ser enviado; a escalabilidade; a mobilidade e desconexões momentâneas.

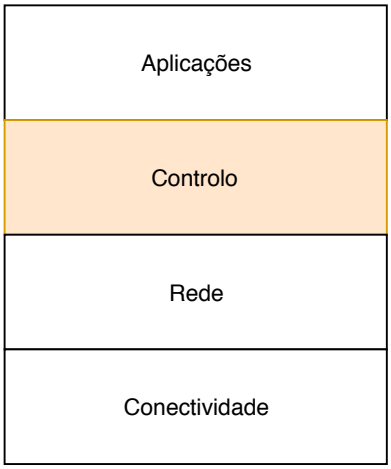

Figura 1.1: Middleware

Para além disso os utilizadores que interagem com o sistema, utilizando os seus dispositivos móveis, devem ter uma boa experiência ao utilizar a plataforma, sendo este outro fator a ter em conta.

A aplicação que recebe o resultado da agregação feita pelo sistema, é totalmente independente da *framework*. Esta apenas recebe um fluxo continuo de dados, comportando-se da forma desejada.

1.3 Proposta de Solução

Este projeto enquadra-se no contexto do projeto "Hyrax" [12] que investiga nuvens de dispositivos na periferia a partir da formação de redes de dispositivos móveis, de forma a permitir a partilha e armazenamento de dados de utilizadores que estão relativamente próximos.

Foi a partir deste projeto de formação de redes de telemóveis que surge a ideia da cooperação entre esses dispositivos. Sendo assim, o objetivo desta dissertação é desenvolver uma *framework* genérica para controlo cooperativo. Esta plataforma permite a vários dispositivos conectarem-se a um servidor de modo a cooperarem entre si com um objetivo comum. A camada de controlo, representada a laranja na Figura 1.1, é o foco desta dissertação, fornecendo assim um serviço que agrega os vários comandos emitidos pelos dispositivos num comando final e único.

Para além desta camada de controlo, foi desenvolvido um protótipo completo, que permite a conectividade, passando pela rede, que agrega os comandos na camada de controlo e que é finalmente usado numa aplicação destino.

Os componentes desta solução são:

Criação de Servidores que suportam a conexão entre os smartphones e a sua aplicação.
Os servidores foram desenvolvidos em linguagem Java [13];

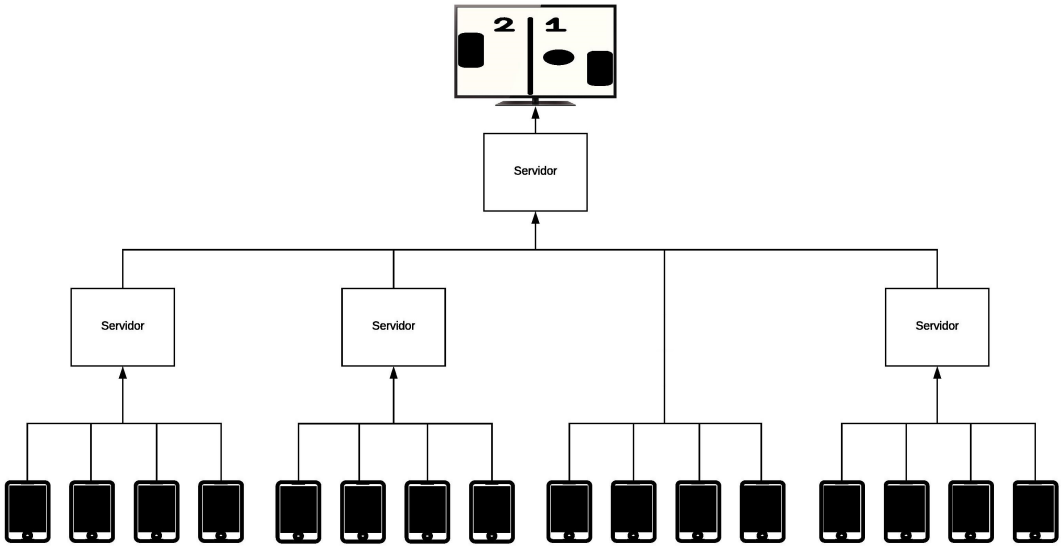

Figura 1.2: Arquitetura da Rede Pretendida

Criação da Aplicação Móvel que permite ao utilizador interagir com o sistema. Esta aplicação foi desenvolvida em linguagem Android [14]. A tecnologia de conexão a utilizar fica ao critério do programador, e não tem impacto na finalidade do projeto.

Os servidores, recebem os comandos provenientes dos dispositivos móveis e têm de os juntar num comando único, seguindo um algoritmo de agregação. Esse algoritmo é independente em relação ao resto do sistema, pois caberá ao programador decidir qual o algoritmo que melhor se adequa à sua aplicação destino.

A proposta de solução é de uma rede em árvore, em que os vários dispositivos móveis comunicam com um ou vários servidores, sendo que estes interagem com o jogo, como está representado na Figura 1.2

1.4 Contribuições

Esta dissertação apresenta as seguintes contribuições:

- Conceção e implementação de uma *framework* que permite a computação do comando final através de vários fluxos de comandos gerado por dispositivos móveis;
- Conceção e implementação de uma aplicação para *smartphones* Android que suporta o envio de informações para o sistema implementado;
- Avaliação da *framework* e aplicação desenvolvidas, em ambiente real e simulado.

1.5 Estrutura do documento

Este documento conta ainda com mais 4 capítulos para além da Introdução.

Tecnologias e Trabalho Relacionado (Capítulo 2) Neste capítulo, são apresentadas as principais áreas de interesse em relação ao contexto desta dissertação, explorando algumas das possíveis tecnologias e *frameworks* a utilizadas.

***Middleware* para Controlo Cooperativo (Capítulo 3)** Este capítulo é dedicado ao sistema desenvolvido e de como este foi implementado. O capítulo inicialmente conta com uma secção dedicada à visão geral de todo o sistema sendo que as secções seguintes abordam os detalhes de implementação dos vários componentes.

Resultados Experimentais (Capítulo 4) Neste capítulo é apresentada uma avaliação do sistema em relação ao seu comportamento. Inicialmente, o capítulo conta com uma descrição das metodologias utilizadas e em seguida são expostos os resultados obtidos em ambiente real e simulado. O capítulo conta ainda com uma descrição do caso de estudo utilizado nos testes.

Conclusões (Capítulo 5) Por fim, o último capítulo apresenta uma conclusão geral desta dissertação e aborda ainda possível trabalho futuro.

Capítulo 2

Tecnologias e Trabalho Relacionado

Neste capítulo são abordadas quais as possíveis tecnologias para a realização desta dissertação, de forma a permitir uma melhor compreensão do funcionamento do sistema e quais as opções disponíveis para a realização do projeto. São também expostos alguns trabalhos realizados relacionados com o tema desta dissertação.

2.1 Controlador

Um controlador é um dispositivo utilizado em consolas ou qualquer outro sistema de entretenimento para interagir com um jogo, geralmente para movermos um objeto ou uma personagem. Os controladores estão geralmente ligados através de um cabo, mas com o avanço da tecnologia já é possível ligar o controlador ao dispositivo que executa o jogo através de tecnologias sem fios.

Os dispositivos existentes podem ser ratos, comandos de vários tipos, teclados, *joysticks*, entre outros. Existem ainda outro tipo de controladores específicos, como volantes para jogos de corridas ou pistolas de luz para jogos de tiros. Exemplo destes controladores, são os os da empresa Sony-*DUALSHOCK 4* [15] ou da empresa Nintendo-*Wii Remote Plus* [16].

O objetivo deste projeto é utilizar um dispositivo móvel, como um *smartphone* ou um *tablet*, como controlador, desempenhando algumas das funcionalidades que são disponibilizadas por controladores já existentes. Alguns destes permitem várias funcionalidades como as que estão descritas na Tabela 2.1. Contudo, o controlador desenvolvido não necessita de implementar todas as funcionalidades dos já existentes. Este tem funcionalidades semelhantes a um *joystick* clássico, com a adição de botões. As suas funcionalidades são:

- o controlo do movimento do jogador através da inclinação do dispositivo, medindo a velocidade com que esta foi feita;

Tabela 2.1: Visão Geral sobre Controladores Existentes

Funcionalidade/Comando	DUALSHOCK 4	Xbox Controller	Wii Remote	Pretendido
Direcionar apenas o jogador	Sim	Sim	Sim	Sim
Direcionar o jogador e a câmara	Sim	Sim	Não	Não
Pressionar Botões	Sim	Sim	Sim	Sim
Pressionar Vários Botões Simultaneamente	Sim	Sim	Sim	Não
Controlar a Intensidade do Movimento	Não	Não	Sim	Sim

- premir um ou mais botões através do toque.

No livro [17], foi feita uma investigação que compara alguns controladores existentes no mercado, em termos de performance, sendo que maior parte dos utilizadores prefere utilizar o rato como controlador, preferindo-o a controladores como o *DualShock 4*. Contudo, dependendo do jogo, há controladores que são mais adequados que outros.

Katzakis et al. [18] desenvolveram um controlador que permite aos seus utilizadores emitirem comandos através do toque e orientação do dispositivo. Este tipo de tecnologia é muito utilizada em aplicações como jogos. Foram várias as técnicas utilizadas neste artigo, contudo, a forma de como um *smartphone* interage com uma aplicação só depende da criatividade do programador. Ficou documentado que a técnica favorita dos utilizadores, entre as técnicas apresentadas, é a de apontar-e-tocar, ou seja, os utilizadores apontam o seu dispositivo para onde querem que o objeto de jogo se desloque e clicam num botão de forma a move-lo nessa direção.

Na aplicação desenvolvida nesta dissertação, as funcionalidades são menos complexas do que as do artigo acima referido, pois apenas é necessário a informação sobre a inclinação do dispositivo e se houve alguma interação com alguns dos botões disponíveis.

2.1.1 Como suportar comandos num dispositivo móvel

Para suportar as funcionalidades os dispositivos têm de conseguir detetar a informação que um utilizador lhe está a transmitir, seja através do toque ou através do movimento do dispositivo, para isso é necessária a utilização de sensores que se encontram presentes na maioria dos *smartphones* e *tablets* utilizados atualmente.

Existe um elevado número de sensores presentes num dispositivo móvel que são utilizados para medir a velocidade, aceleração, a rotação ou a localização. Existem também, em alguns dispositivos mais modernos, outro tipo de sensores como medidores de calor ou de deteção de metais, contudo esses sensores não serão abordados neste projeto.

O Sistema Operativo *Android* [14], contém *frameworks* para interagir com os sensores disponíveis.

Sensores: A maioria dos dispositivos móveis contêm sensores [19] poderosos que podem dar informações como o movimento e orientação. Estes sensores fornecem uma grande quantidade de dados com elevada precisão. Para interagir com os sensores disponibilizados utiliza-se a Classe `SensorManager` disponibilizado pelo Android.

`SensorManager` é uma classe do SO Android que gere todos os sensores de um dispositivo. Neste contexto os sensores a utilizar são: o giroscópio e o acelerómetro [20].

Giroscópio mede a percentagem de rotação em Radianos, em volta dos vários eixos do dispositivo x, y e z. Este tipo de sensor fornece dados sobre a rotação sem qualquer filtragem ou correção de ruído e deriva. A deriva é a soma dos pequenos erros integrados ao longo do tempo. Assim sendo, estes têm de ser compensados, utilizando outro tipo de sensor como o acelerómetro

Acelerómetro mede a aceleração aplicada ao dispositivo, incluindo a força da gravidade. Conceptualmente, o sensor determina a aceleração que é aplicada ao dispositivo (AD) medindo as forças aplicadas ao próprio sensor (Fs). Contudo, a força da gravidade tem sempre influência na medição. Assim sendo o aceleração segue a seguinte relação:

$$AD = -g - (1/mass) \sum Fs$$

Por esta razão, quando o dispositivo está parado (e não está a acelerar), o acelerómetro lê Fs = 9.81 m/s^2. Pela mesma razão, se o dispositivo estiver em queda livre o sistema de forças Fs = 0 m/s^2.

Os acelerómetros usam o sistema de coordenadas *standard*, ou seja, se empurrar a partir da esquerda (movendo-se para a direita), o valor da aceleração x é positiva, ao empurrar para cima, a partir de baixo, o valor de y é positivo.

Ambos os sensores são implementados em hardware.

Botões: Para além dos sensores, o utilizador poderá comunicar com o sistema, através de botões, sendo estes utilizados para interagir com o jogo diretamente, ou indiretamente, assim como aceder a algum tipo de menu. Para isso, é necessária a utilização de ferramentas que permitam a comunicação entre o utilizador e o jogo através de botões. Os botões são disponibilizados pelo SO Android.

2.1.2 Conclusões

O dispositivo móvel é utilizado como o controlador remoto que interage com um jogo. Para isso as suas funcionalidades têm de suportar as necessidades desse jogo, como mudanças de direção e possibilidade de pressionar botões.

A mudança de direção é feita através da inclinação do dispositivo e para isso irão ser usados os sensores descritos na secção 2.1.1. Consoante a sua inclinação, são retirados os valores da sua rotação e são esses os utilizados para definir a direção do objeto no jogo.

Em relação aos botões, a forma mais direta será usar a Classe Button [21]. É a forma mais simples e direta que o SO Android fornece aos seus utilizadores, tendo ainda a vantagem de facilmente se poder acrescentar mais botões, como por exemplo, um botão de menu ou de sair do jogo.

2.2 Dispositivo Móvel como controlador remoto

Nesta secção vamos abordar as várias formas de conectar um dispositivo móvel ao jogo, ou seja, as várias tecnologias disponíveis para fazer a ligação a um servidor e quais os problemas que poderão surgir.

2.2.1 Tecnologia de Comunicação Sem Fios

Tendo em conta o contexto do projeto, o dispositivo móvel terá de realizar operações à distância. Então um controlador remoto pode utilizar várias tecnologias de comunicação sem fios, entre eles o Bluetooth [3] e o Wi-Fi [2]. Neste capítulo irão ser estudadas também as implementações existentes para Android [14] e Java [13].

2.2.1.1 Bluetooth

O Bluetooth [3] é uma tecnologia de conectividade sem fios, de baixa potência, usada para transmitir áudio, transferir dados e transmitir informações entre dispositivos. Existem dois tipos de tecnologias Bluetooth: *Basic Rate/Enhanced Data Rate* (BR/EDR) e *Low Energy* (LE). As diferenças entre as duas tecnologias são: a LE tem uma latência muito menor, a taxa de transferência é muito superior na BR/EDR, sendo que a BR/EDR tem largura de banda suficiente para transmitir dados audio [22]. Para além disso, tal como o nome indica, a LE tem um menor consumo energético.

Existem três tipos de conexão Bluetooth:

Ponto-a-Ponto (1:1): Utilizado para ligações entre dois dispositivos. Este em BR/EDR é otimizado para transferências de áudio e é usado em dispositivos como auscultadores sem fio ou sistema de mãos livres nos automóveis. Quanto ao LE é usado, idealmente, para transferências de dados e para conectar dispositivos como, por exemplo, monitores de saúde;

Multiponto (1:m): Permite a comunicação de um dispositivo a muitos. Esta tipologia disponível está otimizada para a partilha de informações e é idealmente usada para soluções como o fornecimento de pontos de interesse ou a procura de serviços;

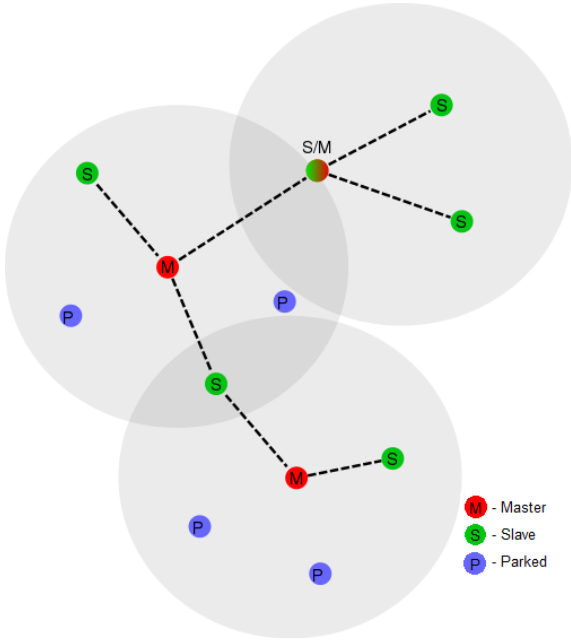

Figura 2.1: Scatternet (Wikipedia[1])

Rede (m:m): É o tipo de conexão que liga muitos dispositivos a muitos. Este tipo de conexão cria uma grande rede de dispositivos que é usada em casos de edifícios com funcionalidades automáticas, redes de sensores ou qualquer outra solução em que seja necessária uma comunicação em rede de forma confiável e segura. Para criar uma rede de dispositivos através da tecnologia Bluetooth, são utilizadas *scatternets*.

Scatternets [23] consiste em dois ou mais *piconets*, como está ilustrado na Figura 2.1. Um *piconet* é um tipo de conexão formado entre vários dispositivos e em que um desses dispositivos é o "mestre"e os outros são "escravos". Sendo assim, com um conjunto de *piconets*, ou seja, com uma *scatternet* (Figura 2.1), é criada uma rede de dispositivos móveis conectados por Bluetooth.

A tecnologia BR/EDR é apenas usado em ligações Ponto-a-Ponto, enquanto nos restantes casos usa-se apenas a tecnologia LE.

A plataforma Android inclui suporte para a tecnologia Bluetooth [24], o que permite que um dispositivo troque dados com outros dispositivos. As funcionalidades do Bluetooth estão disponíveis através das APIs Android Bluetooth. São essas APIs que permitem a conexão entre os vários dispositivos.

Para os dispositivos transmitirem os seus dados entre eles, primeiro têm de ter um canal de comunicação usando um processo de emparelhamento, ou seja, um dispositivo tem de estar disponível e o segundo tem de encontrar o primeiro através de um serviço de descoberta de dispositivos. Depois do dispositivo aceitar o pedido de emparelhamento, podem trocar informações.

[1] https://en.wikipedia.org/wiki/Scatternet

Será necessária também a implementação de um servidor que irá receber os comandos provenientes dos dispositivos, sendo assim, é necessário também o suporte da tecnologia Bluetooth em linguagem Java, pois será essa a utilizada no desenvolvimento do servidor.

Para suportar o desenvolvimento de software habilitado para Bluetooth na plataforma Java, o *Java Community Process* (JCP) [25] definiu o JSR 82, como as APIs Java para tecnologia sem fio Bluetooth (JABWT). O *Bluecove* [26] é uma implementação do JSR 82 e poderá ser utilizada no desenvolvimento deste projeto, tendo em conta que é a implementação mais utilizada, contudo, esta não é oficial.

2.2.1.2 Wi-Fi

O Wi-Fi [2] é uma tecnologia de rede sem fio que usa ondas de rádio para fornecer conexões de Internet. Para ter acesso à Internet através de rede Wi-Fi deve-se estar no raio de ação de um ponto de acesso, e a partir daí será feita a ligação. Tal como o Bluetooth, também permite conexões Ponto-a-Ponto (1:1) ou Multi-ponto (1:m).

As APIs Wi-Fi [27] fornecem um meio pelo qual os dispositivos podem comunicar com o nível inferior que fornece acesso à rede. Quase todas as informações do dispositivo estão disponíveis, incluindo velocidade da conexão, o endereço IP, o estado da rede e ainda informações sobre outras redes que estão disponíveis. Alguns outros recursos da API incluem a capacidade de digitalizar, adicionar, guardar, encerrar e iniciar conexões Wi-Fi.

É usada a classe `java.net.NetworkInterface` [28], para aceder a um dispositivo por Wi-Fi em Java. Esta classe representa o ponto de conexão de rede para os pontos de acesso. Estes pontos fornecem os parâmetros necessários para estabelecer e gerir uma pilha de protocolo TCP/IP. Por norma, as aplicações apenas necessitam de usar a API do ponto de acesso (AccessPoint) para conectar e desconectar. Sendo que a classe `java.net.NetworkInterface` fornece os métodos necessários para fazer a conexão e desconexão aos pontos de acesso. As configurações de rede e o comportamento variam amplamente e dependem de implementações específicas de dispositivos e infraestrutura de rede, dessa forma, as aplicações devem estar preparadas para lidar com falhas e recuperar dessas o melhor possível.

O Wi-Fi Direct [29] é uma variante da tecnologia Wi-Fi que permite que vários dispositivos se conectem sem necessitarem de um ponto de acesso. Telemóveis, câmaras, impressoras, computadores, podem conectar-se e transferir conteúdo de uma forma rápida e fácil. Os dispositivos podem fazer uma conexão de um para um, ou conectarem-se em grupos.

É possível formar redes de dispositivos através da tecnologia Wi-Fi Direct [30] e mesmo combinar tais redes com acesso a infra-estrutura [31]. Estas abordagens podem ser úteis, de forma a reduzir o número de ligações a um servidor.

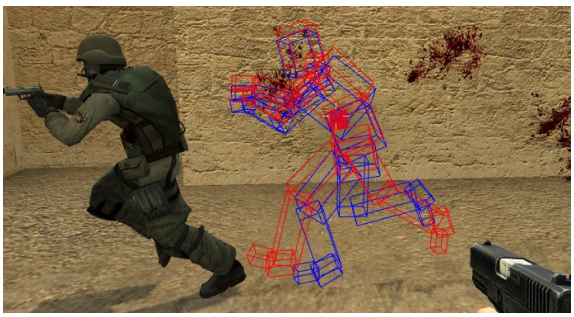

Figura 2.2: Compensação de atraso de Input (Valve [32])

2.2.2 Atrasos no input (Latência)

Devido a fatores como a latência ou limite da largura de banda, pode haver atrasos na transferência de dados, o que se torna num sério problema em jogos online, principalmente nos jogos de velocidade rápida, como é o caso dos jogos de tiros em primeira pessoa. Para que isso não aconteça, tem de haver uma maneira de mascarar esses atrasos de forma a que o utilizador não tenha a noção que estes problemas existem.

Uma das formas de mascarar o atraso de input é com a técnica de previsão [32], ou seja, prever onde é que o jogador irá estar, consoante a sua velocidade e direção, como está representado na Figura 2.2.

Outro problema que surge é o facto de poder receber uma posição nova que está muito adiantada à posição em que se encontra a personagem visualmente, o que pode fazer com que exista pequenos saltos da posição das personagens, isto é, o movimento não é natural, e para resolver este problema existem técnicas que localizam a posição em que a personagem se encontra e move a personagem visualmente durante um curto período de tempo, por vezes, mais rápido que o normal, o que permite alcançar a verdadeira posição da personagem, em vez de fazer o salto nos gráficos. Contudo, isto pode ter problemas a nível visual, por exemplo na colisão com obstáculos.

2.2.3 Granularidade

Os sensores dos *smartphones* são extremamente sensíveis, contudo não são totalmente confiáveis, mas para o caso dos controladores de jogo essa confiabilidade não é assim tão importante, pois o utilizador não sente a necessidade de saber com a precisão máxima, por exemplo, o nível de inclinação, necessitando apenas da sua direção e intensidade. Com isto, o que se pretende dizer, é que é possível mascarar a sensibilidade do dispositivo com a granularidade com que são medidos os valores, ou seja, existe uma diferença significativa de valores para que lhes seja dada importância.

2.2.4 Conclusões

Nesta dissertação desenvolveu-se uma *framework* independente da tecnologia de comunicação. Essa tecnologia será definida pelo programador.

Tendo como base o artigo [33], é possível construir uma rede de dispositivos através do Bluetooth ou Wi-Fi em que os vários comunicam entre si.

No que se refere à latência, o problema é possível ser resolvido utilizando técnicas já existentes, como a técnica da previsão, de modo a que os utilizadores não sintam quaisquer problemas na fluidez do seu jogo.

Quanto à granularidade, fica ao critério do programador decidir qual a diferença de valores entre comandos para lhe que seja dada importância, pois para diferentes aplicações, necessitam de uma diferença de valores diferentes de outras. No exemplo do *Pong*, referido na Secção 1.2, apenas é necessário saber se o jogador pretende direcionar a raquete para a direita ou para esquerda. Num exemplo diferente, como um jogo de corridas de carros, já é necessário saber com precisão a direção que se pretende direcionar o veículo.

2.3 Dispositivo Móvel como controlador cooperativo

Nesta secção são abordadas questões relativamente à forma de como os vários dispositivos irão comunicar simultaneamente com o sistema, sendo que essa comunicação é efetuada através de um ou mais servidores.

O servidor tem como função receber os dados provenientes dos controladores e enviar essa decisão para a aplicação destino. Essa pode ser um jogo ou outro dos exemplos referidos anteriormente.

Este servidor recebe vários fluxos fornecidos pelos controladores, depois une esse conjunto de fluxos num único como está representado na Figura 2.3. É então esse fluxo único que a aplicação destino tem acesso. Consequentemente, surgem várias questões: como o servidor desempenha as suas funções, como resolver problemas relacionados com escalabilidade e mobilidade, entre outras.

2.3.1 Servidor com ou sem estado

Tendo em conta o contexto do projeto, uma reflexão tem de ser feita à volta do estado do servidor, pois vários dispositivos vão ser conectados a um ou mais servidores e estes terão, ou não, necessidade de guardar informações relativamente aos clientes. Ambas as abordagens, com ou sem estado, têm vantagens e desvantagens e deverá ser escolhida a forma que mais se adequa a este projeto.

Servidor com estado guarda a informação relativamente a cada cliente de um pedido para o outro, ou seja, cada cliente ao conectar-se ao servidor terá de iniciar sessão no mesmo.

A sessão é o tempo em que um cliente está conectado ao servidor. Isto traz vantagens, como ser eficiente, assim os clientes não têm de fornecer sempre todas as informações ao fazer uma operação no servidor, isto porque estas ficam guardadas enquanto

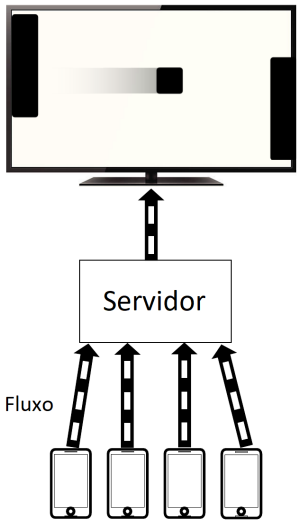

Figura 2.3: Vários Fluxos para um Servidor

a sessão estiver ativa. Desta forma é possível evitar fluxos de dados emitidos pelos clientes, isto é, se a informação que o cliente fornece é exatamente a mesma que a já guardada no estado do servidor, essa informação perde importância. Deste modo, o cliente apenas terá de enviar informações quando estas são relevantes, ou seja, quando são diferentes das anteriores ou quando já passou tempo suficiente para estas serem novamente relevantes. Quando um cliente deixa de fornecer informação, o servidor assume que este cliente deixou de existir e termina a sua sessão.

Por outro lado, também tem desvantagens: a dificuldade em recuperar falhas, pois quando estas existem tem de restaurar todos os estados que tinha guardado; e ainda a quantidade de dados que é possível um servidor guardar. Quando a quantidade de dados for muito grande o servidor poderá ficar sobrecarregado e isso é um problema a ter em atenção.

Servidor sem estado não guarda nenhuma informação quanto aos dados enviados pelo cliente, ou seja, não existe sessão.

Quanto às suas vantagens, será mais simples programar, pois nada fica guardado em memória e apenas se trabalha com os dados fornecidos no momento. Outra vantagem é a facilidade de recuperar de falhas, pois não há estados a restaurar.

Uma das desvantagens referentes a este tipo de servidor é que o cliente tem de enviar todo o seu estado quando envia um pedido, de forma a autenticar-se no servidor. Para além disso, outra desvantagem é que o cliente tem de enviar fluxos de dados contínuos para definir o seu estado, isto é, por exemplo, caso o utilizador queira que a sua personagem num jogo não mude de direção, o cliente tem de constantemente enviar ao servidor que a direção é a mesma.

2.3.1.1 Discussão

Nesta dissertação o servidor escolhido foi um servidor com estado, pois é a forma mais eficiente de resolver os problemas em questão. Assim, não existe necessidade do envio de fluxos constantes, evitando-se a sobrecarga do servidor.

A informação de referente a cada cliente é relativamente pequena, ou seja, é apenas necessário transferir os dados referentes à identificação do utilizador e qual o comando que este pretende efetuar.

É importante salientar que a memória dos servidores é limitada, não podendo ser guardados todos os dados, assim foi necessária a implementação de um tempo de vida sobre a informação fornecida por cada cliente. Após um período de tempo sem receber quaisquer dados, o servidor esquece essa informação, terminando a sessão do mesmo. Posto isto, sempre que haja uma mudança, por exemplo de direção, no lado do cliente, essa informação tem de ser fornecida ao servidor de modo a que este atualize a sua informação.

Sendo assim, o servidor não precisa de guardar todas as informações relativas ao cliente, sendo que apenas é necessário guardar informações relevantes como qual a última informação comunicada pelo cliente e quando foi essa informação fornecida.

O estado do servidor fica apenas em memória por isso não há persistência, logo quando existem falhas, o servidor ao recuperar, inicia sem dados, esperando pela informação dos clientes, isto porque o impacto da mobilidade dos controladores, quando o servidor a que está conectado falha, não é relevante para o desempenho global do sistema.

2.3.2 Escalabilidade

Para que o número de jogadores seja consideravelmente grande, apenas um servidor pode não ser o suficiente, pois um servidor não deve ter demasiados controladores conectados devido a problemas quanto à sua performance.

Por esta razão, é necessária a existência de vários servidores e a sua organização será em árvore, ou seja, cada um destes recebe um conjunto de fluxos de comandos, fornecidos por cada controlador, e emite apenas um fluxo para o que estiver acima dele, como está exemplificado na Figura 2.4.

Para a explicação seguinte, é necessária haver uma noção de nível de servidor, ou seja, cada servidor terá de ter associado o seu nível na pirâmide, pois um servidor só se pode conectar a um servidor com um nível superior ao seu.

Contudo, existe um problema com esta abordagem, pois o servidor de nível máximo (que comunica diretamente com a aplicação destino) torna-se um ponto de falha único.

Um dos problemas que se encontra quanto à plataforma ser extensível é o facto de não se saber *à priori* quantos utilizadores irão participar no jogo, logo os servidores terão de se adaptar as circunstâncias, tal como os próprios dispositivos. Quando um controlador se quer conectar ao jogo, terá de ser decidida a melhor forma de este se conectar a um dos servidores.

Este problema é abordado no artigo [34], em que é utilizado um algoritmo de uma árvore virtual, em que é igualmente distribuído o peso por todos os servidores disponíveis. Os testes demonstraram que este algoritmo obtém melhores resultados quando as cargas são heterogéneas, contudo esta opção só é fiável caso o custo de comunicação seja igual para todos os servidores. No caso desta dissertação, servidores diferentes poderão ter custos de comunicação diferentes, devido à distância a que se encontram esses servidores. Assim sendo, a latência é um fator a ter em conta.

A forma utilizada para jogos de grande escala, é a escolha do servidor com a menor latência, como é demonstrado em [35]. Este artigo descreve a forma de como jogos de grande escala se dividem entre servidores consoante a sua localização geográfica e o seu *ping* a cada servidor.

Sendo assim, para um dispositivo se conectar a um servidor tem de ter em conta dois fatores: a sua localização geográfica e a carga do servidor. Posto isto, uma abordagem possível seria implementar um *load balancing* nos servidores.

A quantidade de níveis de servidores está dependente da quantidade de controladores conectados, ou seja, quanto maior o número de dispositivos será também necessário um maior número de servidores. Se houver um grande número de servidores no primeiro nível, será necessário mais servidores no segundo nível. Tendo em conta que existe mais do que um servidor no segundo nível, terá de existir um nível 3, pois o jogo apenas receberá um e um só fluxo de dados.

Outra abordagem a explorar será utilizar os próprios dispositivos para agrupar os dados, isto é, os dispositivos comunicam entre si utilizando uma tecnologia de rede sem fios e um dos destes comunica com um servidor. Desta forma é possível reduzir o número de servidores físicos.

Mascarar Desconexões Momentâneas e Mobilidade Um dos problemas que poderá surgir no sistema a desenvolver é o problema das desconexões.

No caso desta dissertação, jogam em simultâneo um grande conjunto de jogadores e se um destes se desconectar, o servidor deixa de receber dados provenientes desse jogador e este deixará de ter impacto no jogo. Contudo se o número de jogadores for suficientemente grande, a desconexão de um não terá impacto significativo para que os outros notem diferença no jogo. Para o servidor, quando deixa de obter dados relativos a um dispositivo e tiver expirado o seu tempo de vida, deixa de contar com este para a computação do comando final.

Existe ainda o problema da mobilidade, ou seja, caso um utilizador, por algum motivo, se desconecte e volte a conectar-se noutro servidor que não o mesmo que estava conectado anteriormente. Isto pode acontecer caso um utilizador se desloque, saindo do alcance do servidor em que estava conectado, alcançando um diferente e conectar-se a esse. Se existe mais do que um servidor, é porque o número de utilizadores conectados é suficientemente grande para que haja necessidade de existir mais do que um servidor ativo. Dessa forma, se um dispositivo conectar-se a um servidor imediatamente a seguir a ter-se desconectado

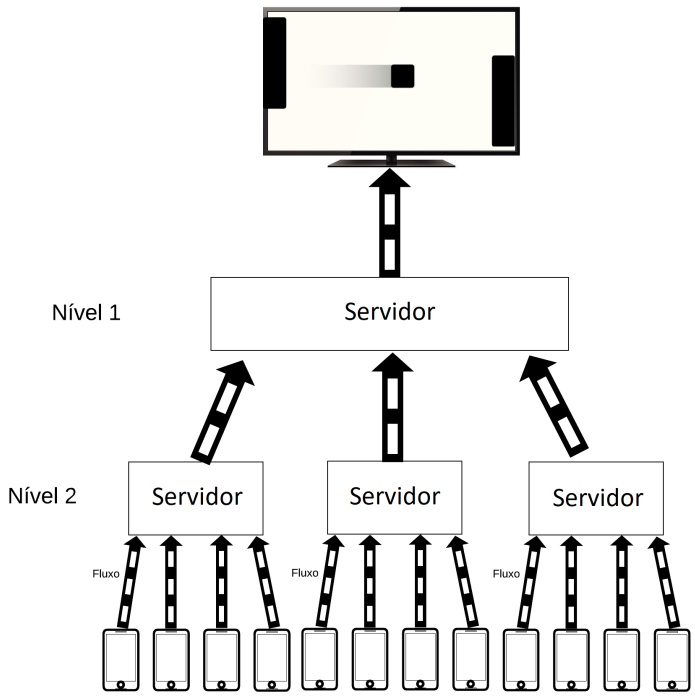

Figura 2.4: Vários fluxos para vários Servidores

do primeiro a que estava ligado, poderá haver duplicação de comandos, pois o dispositivo fornecerá dados ao segundo servidor, mas a sua sessão no primeiro servidor ainda não terminou.

Contudo, isto não é um problema relevante devido ao grande número de dispositivos ligados, pois existe mais do que um servidor. Momentaneamente o jogo terá um comando a mais do que o que devia ter, mas quando o tempo de vida da sessão do dispositivo em relação ao primeiro servidor terminar, o problema ficará resolvido, e as consequências relacionadas com a duplicação de comandos serão mínimas. Se o número de utilizadores for baixo, não existe necessidade de haver mais do que um servidor, logo não existem grandes problemas relacionados com a mobilidade.

Latência Esta questão já tinha sido abordada na Secção 2.2.2, contudo existem diferenças entre resolver problemas de latência quando apenas existe um, e só um, jogador e quando existe uma grande quantidade de jogadores, pois diferenças de milissegundos que têm grande importância quando um jogador que interagir, por exemplo, com uma personagem de um jogo, não têm tanta importância quando uma grande quantidade de jogadores interagem com a mesma personagem, porque o servidor está a receber fluxos constantes de dados fornecidos pelos vários utilizadores e se algum conjunto de dados chegar mais tarde, este é mascarado pelo outros que já se encontram no servidor.

Por outro lado, pode existir problemas relacionados com a latência devido ao número de camadas na árvore de servidores. O tempo que demora um comando proveniente de

um servidor de nível muito baixo, pode ser maior que o desejado, sendo isto um ponto importante quanto à usabilidade da *framework*.

2.3.3 Computação do comando final

O servidor recebe uma grande quantidade de fluxos e terá de computar esses valores de modo a gerar apenas um conjunto de dados para enviar para o jogo, mas coloca-se a questão de como é que esses se irão unir num único fluxo. Para que isso aconteça é necessário utilizar algum tipo de estratégia de modo a unir os fluxos.

Existem problemas a nível de servidor, pois, segundo a Figura 2.4, os servidores do nível 2 irão unir-se no nível 1, mas se um dos servidores contiver um maior número de dispositivos, terá de ser dada maior importância a esse, por isso, para além do resultado calculado pelo servidor do nível 2, tem ainda de passar para o nível acima a informação relativa ao número de dispositivos que compuseram aquele fluxo. Existe então a questão do que se transmite entre níveis, e qual será a dependência de cada algoritmo dessa transmissão.

Esta questão da agregação de comandos vindos de múltiplos nós ligados a um único servidor é abordada no artigo [36]. Neste são apresentadas várias estratégias de agregação e sua a comparação em termos de performance.

Um algoritmo possível e relativamente simples de utilização será um algoritmo de agregação, ou seja, fazer a média dos valores recebidos, por exemplo, num caso com dois dispositivos, se um dos dispositivos diz ao servidor que quer rodar 6 unidades para a esquerda e o se o outro pretender rodar apenas 2 unidades, a resposta que o servidor fornece ao jogo é que os utilizadores pretendem rodar 4 unidades para a esquerda. Esta é uma forma relativamente simples e eficaz da aplicação dar importância a todos os utilizadores de igual forma. Caso haja mais que um nível de servidores é necessária a transmissão do comando emitido, mas também o número de dispositivos conectados, de forma a ser calculada a média ponderada.

Podemos comparar os algoritmos a usar com o modelo de programação *MapReduce* [37]. O *MapReduce* foi desenhado para processar um grande volume de dados em paralelo, dividindo o trabalho por um conjunto de nós independentes. Depois cada nó computa esses dados e são combinados por um nó mestre.

Nesta dissertação, acontecerá algo semelhante em que os vários servidores computarão os comandos recebidos e posteriormente, enviam esses comandos para o servidor principal, sendo este que combina esses valores, obtendo um valor final.

Posto isto, os algoritmos a utilizar têm de ser operações comutativas e associativas, tal como acontece com as reduções no *MapReduce* [37].

Outras estratégias a abordar seria, por exemplo, atribuir diferentes pesos aos vários dispositivos, sendo que os que teríam maior peso, teríam também maior impacto no jogo. Este dispositivo de maior peso podería variar em função do tempo, dando assim importância a vários utilizadores. Contudo esta abordagem retira alguma importância aos

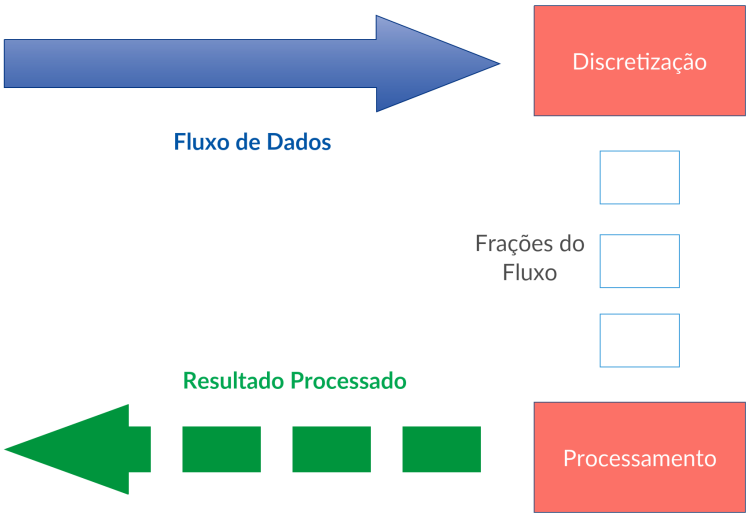

Figura 2.5: Discretização do Fluxo

utilizadores secundários. Também é possível utilizar algum algorítmo de aprendizagem em que o algoritmo dá importância a cada dispositivo consoante as necessidades, isto é, existir um consenso de qual o melhor comando a ser executado. Este último método é abordado em [38]. Neste artigo, são comparados vários algoritmos de consenso e em que situações se adequa cada um dessas estratégias. Nesta dissertação, não é necessário um algoritmo com complexidade muito elevada, tendo em conta que o algoritmo é independente da *framework* a ser desenvolvida. O algoritmo a utilizar no protótipo desenvolvido é apenas usado para testes de validação.

Outras questões relacionadas com o comando final podem surgir, uma delas será a necessidade de discretizar o fluxo de informação proveniente dos dispositivos e se existe a necessidade de ter em consideração um histórico dos recentes comandos enviados. Existem métodos de discretização como está representado no artigo [39] em que é possível obter os vários comandos ao longo do tempo, como está representado na Figura 2.5. Nesta dissertação, o método de discretização a ter em conta é o referido no artigo [39] *Equal Width discretization* (EWD), em que consiste em dividir um fluxo de dados em k fragmentos iguais de dados.

Contudo existem outras formas de discretizar o *stream*, como ter uma janela deslizante que percorre o fluxo, como está representado na Figura 2.6, e desliza tendo em conta os comandos anteriores. Este processo é, de acordo com o trabalho reportado no artigo [40], utilizado no requisito número 2: usar SQL em fluxos, dos 8 para o processamento de fluxos de dados. Esta estratégia é utilizada em vários sistemas de processamento de streams, como o Microsoft Azure Stream Analytics [41]. No artigo [40], são ainda reportados requisitos importantes, como:

- os dados estarem sempre em movimento, guardando pouca informação de modo a obter pouca latência;

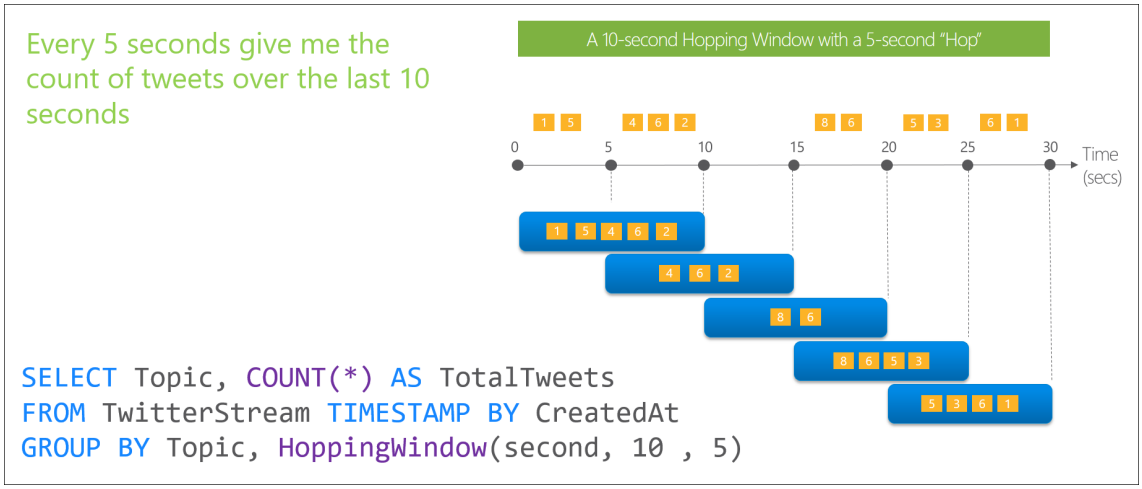

Figura 2.6: Janela deslizante (Microsoft[2])

- tratar de maneira eficaz as imperfeições do fluxo, como atrasos ou a ordem errada de dados;
- gerar resultados previsíveis;
- integrar os dados já guardados com os novos dados recebidos, isto é, depois do *stream* estar discretizado será necessário ter guardar o histórico com os últimos comandos lidos de modo a evitar mudanças abruptas de comandos, lendo o próximo segmento do fluxo tendo em conta o que já foi lido;
- processar e responder instantaneamente aos pedidos, de forma a que tudo decorra com fluidez.

2.3.4 Conclusões

Concluí-se neste capítulo que o servidor ou servidores têm estado, contudo não precisam de guardar toda a informação sobre os clientes, apenas a necessária para a computação do comando final.

O número de servidores está dependente do número de jogadores, então é necessário um número suficiente de servidores de forma a suportar todos os comandos recebidos, podendo-se ainda recorrer a técnicas que permitam um alívio de carga como a utilização de dispositivos como servidores.

Quanto aos problemas relacionados com as desconexões dos utilizadores são razoavelmente fáceis de resolver, isto se o número de dispositivos conectados for suficientemente grande. Caso contrário a desconexão e mobilidade, num ambiente com poucos intervenientes será difícil de mascarar e poderá tornar-se percetível aos restantes utilizadores.

[2]https://docs.microsoft.com/en-us/azure/stream-analytics/media/stream-analytics-window-functions/stream-analytics-window-functions-hopping-intro.png

Por fim, surge a melhor forma de como computar o comando final. É aqui que surgem os maiores desafios, pois tem de existir estratégias como a de janela deslizante para retirar todas as informações de um fluxo, de uma maneira suave, para que lhe possa ser, juntamente com os outros fluxos, aplicado o algoritmo de decisão. O algoritmo a utilizar para gerar o comando final é independente do sistema e será definido pelo programador, recebendo apenas um conjunto de informações já filtradas.

Também é preciso ter em conta o que é transmitido entre níveis de servidores, pois existem algoritmos que precisam de mais informações do que outros.

Capítulo 3

Middleware para Controlo Cooperativo

Neste capítulo é apresentado em detalhe o sistema implementado. Na primeira secção é apresentada uma visão geral do sistema sendo que nas secções seguintes são explicados os vários componentes do sistema.

3.1 Visão Geral

A solução desenvolvida nesta dissertação é um *middleware* para controlo cooperativo, ou seja, foi concebida uma *framework* genérica que permite a cooperação entre dispositivos móveis de modo a obter objetivos em comum.

O sistema combina os comandos emitidos pelos dispositivos móveis, doravante denominados controladores, num único comando a ser utilizado numa aplicação destino.

Para providenciar a sua funcionalidade, o sistema é composto por múltiplos componentes distribuídos fisicamente, divididos em três elementos principais, sendo eles: controladores, servidores e aplicação destino. Estes elementos estão evidenciados na Figura 3.1.

Os dispositivos têm como função de fornecer os fluxos de comandos ao sistema. Esses fluxos são compostos por vários comandos, podem ser emitidos através de movimentos ou de botões, dependendo da finalidade da aplicação. Os controladores serão móveis, podendo desconectar-se e conectar-se sempre que assim o desejarem, não havendo quaisquer restrições a nível de mobilidade. Os controladores são completamente independentes, não necessitando de qualquer tipo de coordenação entre eles.

O sistema tem como base o servidor de nível 1, como está representado na Figura 3.1. Os servidores têm como função combinar os fluxos emitidos pelos controladores num único fluxo de comandos a ser enviado para a aplicação destino. A forma de como são combinados os fluxos fica ao critério do programador que utilize este sistema, tendo para

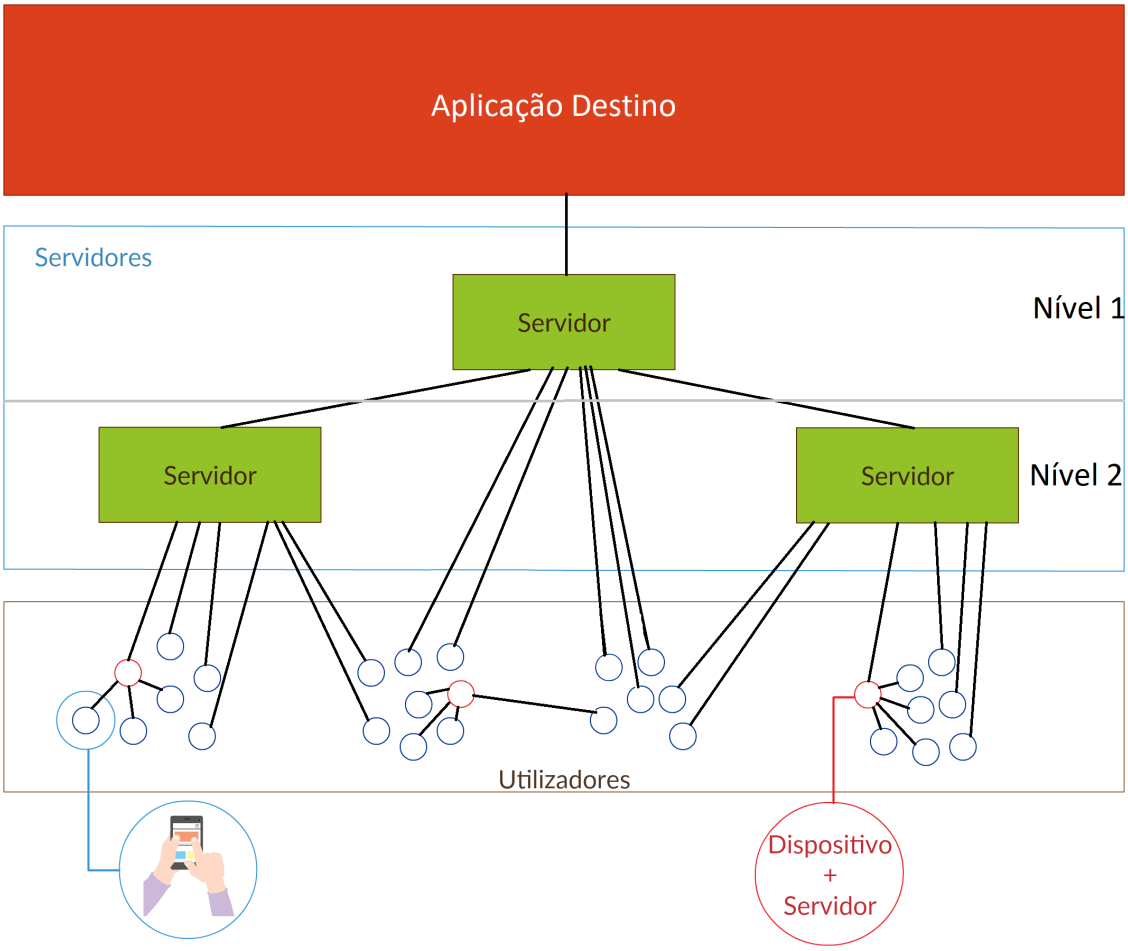

Figura 3.1: Arquitetura geral

tal especificar essa informação à *framework*. Para isso, este tem de fornecer algoritmos para que a agregação de comandos seja feita de acordo com os esses. O resultado da agregação, é posteriormente enviado para a aplicação destino. O servidor principal (nível 1) tem ainda de ser configurado com a frequência com que os comandos devem ser entregues à aplicação. A aplicação é independente do sistema e vice-versa. O servidor de nível 1 apenas comunica o resultado da combinação dos fluxos, consoante a frequência desejada.

Em alguns casos de uso, um servidor pode não ser o suficiente para suportar o número de controladores conectados, e, neste caso, será necessário um maior número de servidores que comunicam o seu resultado com o servidor de nível imediatamente acima.

A forma de comunicação entre controladores e servidores também tem de ser especificada pelo programador e essa comunicação pode ser feita através de qualquer uma das tecnologias apresentadas na Secção 2.2.1, contudo a comunicação entre controladores e servidores é independente do *middleware*, pois o comportamento do sistema será o mesmo independentemente da tecnologia de comunicação utilizada. No entanto, a tecnologia de ligação pode ter impacto na latência das operações.

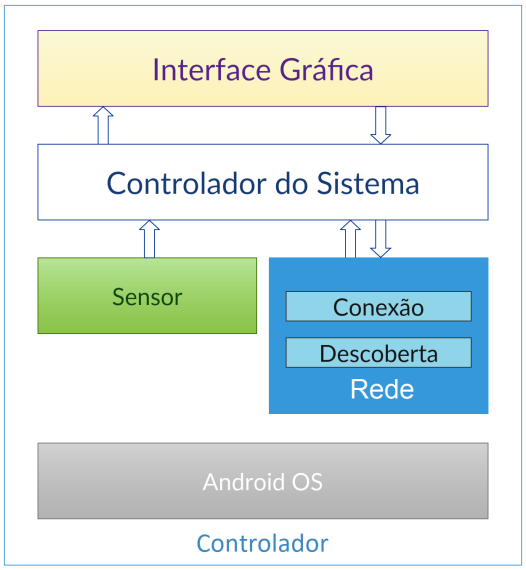

Figura 3.2: Arquitetura Controlador

3.2 Controlador

O controlador é o meio de interação entre os utilizadores e o sistema desenvolvido. A função principal destes é fornecer fluxos de dados ao servidor para que estes possam ser processados e mais tarde chegarem à aplicação destino.

3.2.1 Arquitetura Interna

A arquitetura de controlador divide-se em quatro camadas, sendo elas:

Rede - tem a função de enviar e receber informações do sistema;

Controlo - tem como finalidade receber dados provenientes dos sensores e da interface gráfica e converte-os em comandos válidos e fornece-os à camada de rede. Recebe também informações provenientes da camada de rede que são enviadas pelo servidor. Tem ainda a função de comunicar com a interface gráfica informações sobre a aplicação, como, por exemplo, a inclinação do controlador

Sensor - deteta as ações do utilizador, como a inclinação do dispositivo;

Interface Gráfica - tem como principal objetivo fornecer as informações ao utilizador. Comunica ainda com o controlador do sistema caso o utilizador pressione algum botão disponível

Estas camadas estão apresentadas na Figura 3.2.

3.2.2 Conexão ao servidor

Os controladores verificam quais os servidores disponíveis e faz a conexão a um deles. Caso um controlador esteja no alcance de mais do que um servidor, irá sempre conectar-se ao servidor menos cheio, pois desta forma a carga de cada controlador nos servidores será distribuída de forma igual.

Uma vez a conexão estabelecida, o servidor envia informações sobre a aplicação destino, por exemplo, qual a aplicação, no caso de um jogo, quais as equipas disponíveis. Contudo estas informações mudam consoante a finalidade, ou seja, existem aplicações mais exigentes que necessitam de um maior número de definições a serem estabelecidas. Exemplificando, no caso de estudo utilizado, apresentado na Secção 4.2, as informações a serem transmitidas aos controladores são o nome do jogo a que está a jogar, ou seja: Pong, o número de equipas disponíveis: 2 e caso exista, o nome das equipas.

Para suportar as diferentes exigências de cada aplicação, o sistema envia para os controlador um array de bytes com as características da aplicação destino. A *framework* obriga a implementação de um serializador de bytes, pois, uma vez o serializador implementado, a *framework* sabe construir e desconstruir objetos do tipo requerido.

O canal de comunicação entre o controlador e servidor transfere ainda outro tipo de informações como, sinais para abrandar ou aumentar a emissão de comandos. Existem casos em que o servidor não está a conseguir processar os dados dentro do tempo de processamento pedido, então estes enviam um sinal para os controladores de forma a que estes abrandem a sua emissão de comandos, diminuindo assim a carga na rede. Existe também o caso oposto, pois existem situações em que os servidores processam os comandos rapidamente e têm capacidade para processar mais informações e nestes casos o servidor informa os controladores para estes aumentarem a sua cadência de comandos, caso esta tenha sido diminuída anteriormente, isto é, inicialmente a frequência de comandos enviados pelos controladores é máxima.

3.2.3 Deteção e emissão dos comandos do utilizador

O utilizador pode emitir comandos inclinando o seu controlador na direção desejada. Para detetar qual a direção e qual a intensidade desse movimento, podemos tirar partido dos sensores disponíveis nos dispositivos móveis. Outra forma de emitir comandos é através dos botões presentes na interface gráfica (Secção 2.1.1).

Finalmente, esses valores provenientes dos sensores ou dos botões são convertidos em comandos válidos a ser enviados. A representação de um comando no sistema é dada por:

Identificador do emissor: é o identificador de quem o envia, seja ele um controlador ou um servidor intermédio. Este identificador é gerado através do método disponibilizado pelo *Android* `UUID.randomUUID()`.

Tipo de comando: os comandos têm de ter a identificação do seu contexto, o tipo de comando pode ser, por exemplo, movimento horizontal, botão de salto ou valor de um

servidor. Os tipos de comando variam consoante as necessidades da aplicação destino e o controlador também terá de ter em conta estas informações. Estes comandos são definidos pelo programador utilizando a *framework*.

Valor do comando: valor a processar pelo servidor. Por exemplo, no caso em que o tipo de comando é de movimento horizontal o valor do comando é o valor emitido, em graus, pelo sensor do controlador.

Peso: associado a um comando terá de vir associado um peso. Isto aplica-se mais em casos em que os comandos provêm de outros servidores. Um controlador normal, emite comandos com peso 1, enquanto um servidor emitirá comandos com um peso igual ao número de controladores que lhe estão conectados.

Histórico: nem toda a informação enviada é armazenada no histórico de comandos do servidor, que será abordada mais à frente na Secção 3.3.4.1. Exemplificando, num caso em que hajam dois tipos de comandos, sendo eles "movimento"e "disparo"apenas no primeiro caso faz sentido guardar a sua informação. Isto porque, um movimento pode ser uma direção em graus e caso o utilizador não altere a direção do seu dispositivo, pois não tem a intenção de a mudar, o servidor não recebe a informação, porque não houve uma mudança de comando. Por outro lado, em casos como o "disparo", não quer dizer que o utilizador deseja continuamente disparar. Para evitar estes casos, os comandos enviados têm este componente de forma a permitir ao programador decidir quais os comandos que devem ser ou não armazenados em histórico. No protótipo desenvolvido, o histórico é definido pelo controlador, contudo uma abordagem válida seria ser o servidor controlar esta codificação, ou seja, o servidor decidir quais os comandos que devem ser guardados em histórico, consoante o seu tipo, libertando assim carga na rede.

Outro: é dada a opção de se passar mais alguma informação, opcionalmente, caso haja essa necessidade. No protótipo desenvolvido, utiliza-se esta opção para transferir para o sistema qual a equipa escolhida.

Contudo, nem todos os movimentos feitos pelos utilizadores são do interesse da aplicação final, pois nem todos têm uma diferença significativa do comando anterior de modo a ter importância no sistema: esta é a questão da **granularidade**, já abordada na Secção 2.2.3. Por outro lado, se o utilizador pretender emitir sempre o mesmo comando, ou seja, não quiser mudar, por exemplo, a inclinação do seu controlador, o controlador emite um comando apenas com a informação que está vivo de modo a não sobrecarregar a rede com comandos iguais. O servidor apenas atualiza as informações referentes a este controlador.

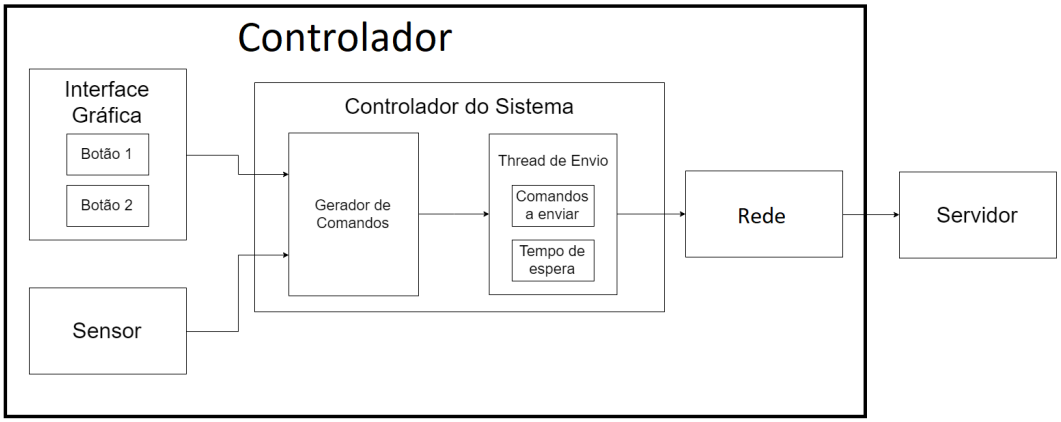

Figura 3.3: Funcionamento de um controlador

3.2.3.1 Granularidade

A granularidade, consiste na diferença significativa de valores, para que seja dada importância suficiente a um comando. O valor da granularidade, ou seja, dessa diferença de valores é parametrizado pelo programador e quanto maior for esse valor, menor será o número de comandos enviados, pois será necessário um movimento mais brusco por parte do utilizador.

Este valor deve ser parametrizado pelo programador consoante as necessidades da aplicação final, pois diferentes aplicações necessitam de níveis de precisão diferente em relação aos comandos que pretende receber.

3.2.4 Envio de dados

O controlador conta com uma *Thread* dedicada ao envio de dados, à semelhança do servidor, que será abordado na Secção 3.3. Essa *Thread* tem como função enviar os comandos para o sistema consoante o tempo definido por este. Exemplificando com a Figura 3.3, o controlador do sistema do dispositivo recebe as informações vindas dos sensores e da interface gráfica e coloca esses valores numa variável de comandos a enviar. Consoante o tempo de espera, este envia os comandos sempre que haja oportunidade. Caso haja comandos novos enquanto a *Thread* está em espera, são atualizados os valores a enviar.

3.2.5 Detalhes de implementação

Existem alguns detalhes na implementação do controlador que devem ser explicadas de modo a uma melhor compreensão do seu funcionamento.

O protótipo de controlador foi desenvolvido no sistema operativo Android [14].

3.2.5.1 Protocol Buffers

Existem várias formas de serialização de dados de modo a permitir o seu envio pela rede. A forma utilizada neste projeto foi a utilização do mecanismo de serialização *Protocol*

Listagem 3.1: Formato da mensagem enviada por um controlador

```
message Command {
    required string id = 1;
    required string type = 2;
    required float value = 3;
    required float weight = 4;
    required bool history = 5;
    optional bytes other = 6;
}
```

Buffer [42]. Este foi o mecanismo escolhido pois ao analisarmos o artigo [43], concluiu-se que este é o mais indicado para este sistema. O *Protocol Buffer* apresenta uma velocidade de propagação bastante rápido e produz mensagens bastante pequenas e é bastante fácil definir os dados a transferir.

Um exemplo de formato das mensagens transferidas é a Listagem 3.1, que representa um comando enviado por um controlador para o sistema.

3.2.5.2 Tecnologia de Comunicação

A aplicação desenvolvida comunica com os servidores através de qualquer uma das tecnologias de comunicação existentes como o Wi-Fi ou Bluetooth, como já foi referido na Secção 2.2.1. A escolha da tecnologia a utilizar fica ao critério do programador que use esta plataforma, adaptando-se às necessidades do contexto da aplicação.

No protótipo desenvolvido a tecnologia utilizada foi o Wi-Fi.

3.2.5.3 Deteção de comandos

O sensor utilizado na aplicação desenvolvida é o TYPE GAME ROTATION VECTOR [44], que fornece uma matriz com a orientação do dispositivo, sempre que há um movimento. É traduzido essa matriz para valores x,y,z e é enviado o valor pretendido para o controlo.

É possível a utilização de mais do que um sensor, contudo na aplicação desenvolvida não existiu essa necessidade, sendo assim, apenas é calculada a direção em que o dispositivo se encontra, em graus, sendo que 90º é a posição do dispositivo na horizontal.

3.3 Servidor

Os servidores são o elemento principal desta dissertação e tem como principal função unir os fluxos enviados pelos controladores. Para que o sistema tenha o comportamento esperado, um fluxo de dados passa por várias fases até chegar à aplicação destino. O fluxo depois de recebido é descretizado e a esse segmento de comandos, são lhes aplicados algoritmos, fornecidos pelo programador e independentes do sistema, de forma a uni-los para que possam ser utilizados.

O servidor principal, que tem como função fornecer os resultados à aplicação destino. Isto significa que pode existir mais do que um servidor, pois para que este projeto seja escalável, pode ser necessário distribuir a carga por vários servidores. Assim sendo, é necessário introduzir o conceito de nível, já abordado na secção 2.3.3. Um servidor tem de lhe ter associado um nível, sendo que o nível 1 será sempre o servidor final, ou seja, o servidor que emite comandos para a aplicação, tal como está representado na Figura 3.1. Desta forma é possível estruturar uma árvore de servidores, de modo a tornar o projeto escalável.

Existem alguns problemas associados à arquitetura em árvore, pois caso apenas exista apenas um servidor, existe apenas um ponto de falha e caso este servidor falhe perde-se toda a informação do sistema e para que tudo volte a funcionar, a única solução é esperar que o servidor volte ao ativo e quando isso acontecer os controladores têm de se conectar novamente ao sistema.

Caso exista mais que um servidor, caso seja o servidor de nível 1 a falhar, o problema mantém-se e todo o sistema tem de esperar que o servidor volte ao ativo, mas caso seja um servidor com nível superior a 1, a aplicação não ira deixar de receber informações. Contudo, os controladores que estão conectados ao servidor que falhe deixam de fornecer informações. Caso hajam muitos servidores no sistema, os controladores que se desconectaram, devido à falha do seu servidor, podem ser divididos pelos restantes servidores, caso só hajam dois servidores e o servidor de nível 1 esteja no seu limite, a única abordagem possível é esperar que o servidor volte ao ativo.

Todas as informações são guardadas em memória e não existe replicação de dados e por isso caso um servidor falhe, essa informação é perdida.

O servidor de nível 1, deve ter sempre controladores conectados, de forma a não desperdiçar recursos, pois uma abordagem em que o servidor do topo da hierarquia apenas junta comandos vindos de outros servidores, não fará qualquer tipo de processamento. Mesmo que um servidor de nível 1 tenha conectado milhares de servidores de nível intermédio, os comandos vindos desses servidores já estão processados, e por isso o tempo de processamento desses comandos é desprezável. Este assunto será discutido com mais detalhe na Secção 4.5.

3.3.1 Arquitetura Interna

A arquitetura de um servidor divide-se em três componentes principais sendo elas:

Camada de rede - tem a função de receber os fluxos e os disponibilizar ao sistema o e fornecer as informações ao nível acima, caso o servidor esteja no nível 1, fornece os dados à aplicação;

Controlador de fluxos - que guarda a informação sobre cada controlador e fornece os fluxos ao controlador do sistema;

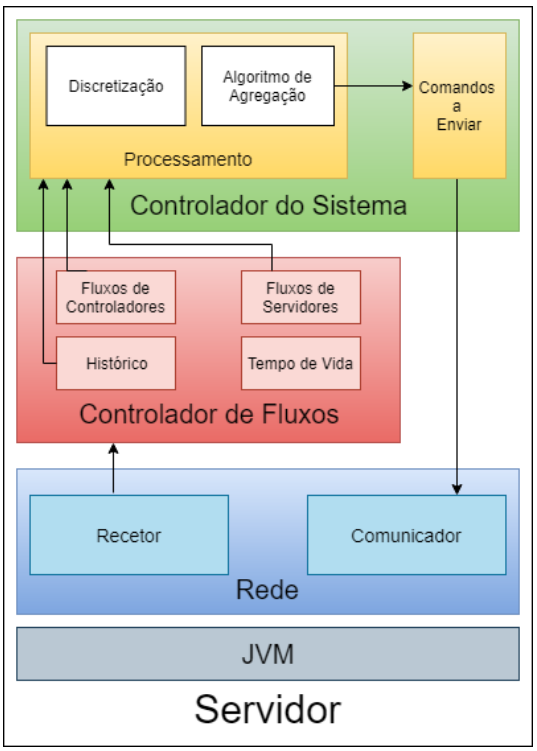

Figura 3.4: Proposta de arquitetura para um Servidor

Controlador do sistema - tem como função processar os fluxos, ou seja, uni-los num único.

Estas camadas estão representadas na Figura 3.4, sendo que a sua composição interna será detalhada ao longo das próximas secções.

3.3.2 Framework

O programador que utilize esta *framework* tem de definir uma série de parâmetros iniciais para o que o sistema se comporte da forma desejada. Esses parâmetros são definidos manualmente.

Em primeiro lugar, tem de definir qual a aplicação final e esta tem de implementar uma interface com um único método **void send(R result)**. Desta forma a *Thread* que envia os resultados para a aplicação, apenas chama este método e o programador fica responsável pela implementação do método, ou seja, a forma de como o resultado chega à aplicação destino. Depois o programador tem de fornecer um mapa de algoritmos em que a sua chave é o tipo de comando. Desta forma, o sistema associa cada tipo de comando a um algoritmo diferente e o programador utiliza essa informação da forma adequada à sua aplicação. Tem de implementar ainda, uma interface ***Connection***, para que possa definir qual a tecnologia de comunicação desejada.

Têm de ser ainda definidos alguns parâmetros como a frequência com que deve ser enviados resultados para a aplicação (em milissegundos) e outros menos importantes,

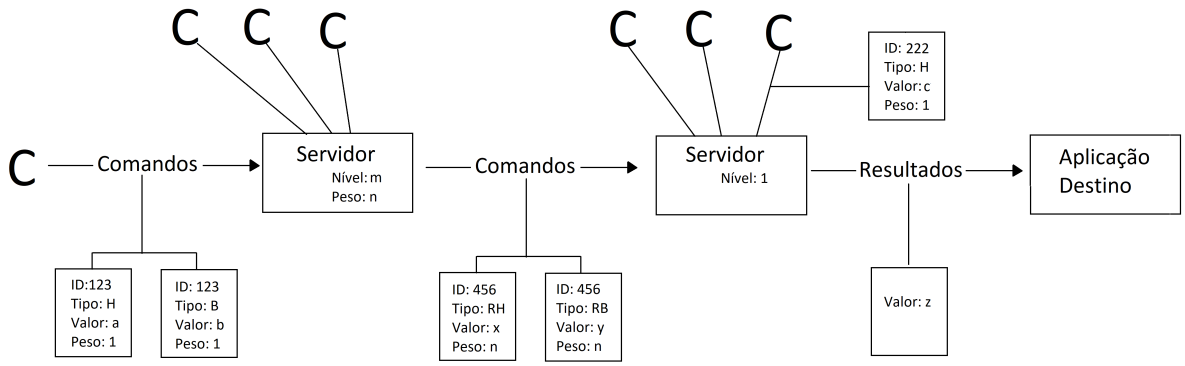

Figura 3.5: Esquema do Sistema desenvolvido

como o nome da aplicação. A *framework* conta ainda com um parâmetro opcional, para que o programador possa definir alguns que não são comuns a todas as aplicações como por exemplo, número de equipas, nome das equipas, entre outros.

Um exemplo da especialização da *framework* é apresentado no Apêndice A.

3.3.3 Conexão dos Controladores

Os servidores são responsáveis por juntar os comandos vindas dos vários controladores.

Sempre que um servidor recebe um comando associa-lhe um tempo de vida, ficando associado ao seu identificador. Desta forma, é possível avaliar a última vez que o controlador ou servidor, enviou comandos. Esse tempo de vida é atualizado sempre que haja novas informações com o mesmo identificador. Caso o servidor não receba qualquer informação dentro desse tempo de vida, remove esse controlador, aliviando assim carga no servidor.

Os fluxos de comandos a serem recebidos têm algumas diferenças, pois fluxos de comandos vindos de um servidor não necessitam de passar pelos mesmos processos que um fluxo normal de emitido por um controlador. Essas diferenças estão relacionadas com o tipo de comandos que geram o fluxo. Ou seja, o tipo de comando enviado por um controlador é diferente do tipo de comando enviado por um servidor, tal como o seu peso. Um controlador tem peso 1, enquanto um servidor tem o peso igual ao número de controladores que lhe estão conectados.

Exemplificando, na Figura 3.5 é apresentado um esquema do sistema proposto, em que os controladores comunicam com um servidor e emitem comandos de vários tipos, ilustrados na figura com o tipo H e B e peso 1, enquanto os servidores de nível diferente de 1, emitem comandos de tipo RH e RB, que significa resultado do processamento de comandos de tipo H e resultados do processamento de comandos de tipo B, respetivamente. Cada servidor terá o peso do número de controladores conectados a este. Na Figura 3.5 esse peso está ilustrado com um peso "n", que varia consoante o número de controladores.

3.3.4 Processamento da Informação

O processamento da informação é fase onde são combinados os comandos.

Os comandos recebidos passam por uma série de processos de modo a que reste apenas um comando único final a ser posteriormente emitido para o nível acima.

Cada servidor tem de lidar com um conjunto não limitado de dados, ou seja, fluxos, que são enviados por um conjunto dinâmico de geradores. Tendo isto em conta, o sistema não sabe ao certo quantos controladores estarão conectados nem quantos comandos serão emitidos. Posto isto, enquanto o servidor recebe esse fluxo continuo de dados, tem de ir produzindo um *output* de comandos para o nível superior com uma cadência que alimente a aplicação final com a frequência desejada.

Tendo isto em conta, o servidor não pode processar o fluxo inteiro de dados de uma vez porque como o fluxo não tem um limite conhecido, não se sabe à partida o seu tamanho. Posto isto, o fluxo de *input* tem de ser dividido em segmentos, ou seja, tem de ser discretizado, de modo a melhorar o tempo de processamento.

3.3.4.1 Histórico

O sistema conta com um histórico de informação de modo a auxiliar o processamento da informação. Cada comando recebido conta com um valor booleano que define se esse comando tem importância suficiente para ser guardado ou não no histórico. Caso um comando tenha essa importância, é armazenado pelo servidor em memória num mapa de comandos em que a chave é o identificador do controlador. Sempre que haja um novo comandos que deve ser armazenado no histórico, substitui-se o valor antigo pelo novo comando.

Desta forma é possível fazer a separação entre comandos que devem ser contínuos, como por exemplo direções, de comandos esporádicos como disparos ou saltos.

3.3.4.2 Discretização

A discretização do conjunto de comandos, abordada na Secção 2.3.3, consiste na separação destes consoante um intervalo de tempo. Isto é, dividir o conjunto de comandos em pequenos segmentos de forma a serem mais facilmente processados.

Esses segmentos são compostos pelos comandos recebidos no último intervalo de tempo, contudo nesse segmento podem estar, ou não, comandos emitidos por todos os controladores ligados ao servidor. Então, para cada segmento de informação o sistema verifica se existe algum identificador no histórico que não esteja nesse segmento. Caso esteja algum em falta, esse comando é adicionado ao segmento.

O intervalo de tempo que segmenta o fluxo deve ser calculado tendo em conta a carga da informação na rede, contudo na implementação desenvolvida, este fator não foi tido em conta, sendo que o intervalo de tempo mantém-se fixo.

Depois da lista de comandos estar completa, ou seja, os novos valores, mais os valores antigos dos controladores que não forneceram novas informações neste segmento de tempo, são lhe aplicados os algoritmos de agregação.

3.3.4.3 Algoritmo de Agregação

O algoritmo é definido pelo programador e independente de todo o restante sistema. A interface fornecida, conta apenas com um método:

Pair<R,Integer> process(CommandList<Command<T> > cList, CommandList<Command<R> > sList, int weight) O método recebe uma lista de objetos do tipo T, cList, que representa os comandos provenientes dos controladores conectados ao servidor e uma lista de comandos de tipo R, que representa os comandos enviados pelos servidores do nível abaixo. Esta distinção é feita, pois apenas a lista de comandos enviada pelos controladores necessita de ser processada, enquanto a outra que o método recebe, sList, são resultados emitidos por outros servidores, ou seja, já processados, que apenas precisam de ser somados ao resultado do processamento. Conta ainda com um parâmetro *weight* que representa o número de controladores conectados ao servidor.

Este objeto do tipo CommandList é fornecido pela API e tem as seguintes funcionalidades:

CommandList<Command<T> > removeDuplicates() que devolve a mesma CommandList<Command<T> >, mas sem dados repetidos, isto é, um controlador tem apenas um e um só comando neste segmento de dados.

CommandList<Command<T> > smoothContent() que devolve também a mesma CommandList<Command<T> >, mas com valores mais perto dos que eram esperados. Isto pode ser utilizados em casos em que hajam utilizadores que tenham emitido comandos completamente diferentes da maioria.

O método *process()*, devolve um par, com um valor de tipo genérico R e com um peso associado. Desta forma o programador pode dar pesos diferentes a tipos de comando diferentes, exemplificando, se observarmos o caso do jogo do *Pong* os movimentos são contínuos, por isso, existe sempre a necessidade de saber quantos controladores desejam mover para um lado ou outro. Em casos de jogos como o *Arkanoid* (descrito na Secção 1.2) cabe ao programador tratar desses casos da forma que mais se adeque à sua aplicação.

3.3.5 Detalhes de implementação

O servidor foi desenvolvido em linguagem Java [13] e divide-se em três partes principais sendo elas: receção, processamento e envio de comandos.

3.3.5.1 Receção de Comandos

Esta plataforma conta com uma *Thread* dedicada à receção de comandos.

Para a receção de comandos o sistema utiliza uma classe disponibilizada pelo API do Java NIO: a classe *Selector*. O *Selector* fornece um mecanismo em que é possível monitorizar vários canais de comunicação e deteta quando é que estes canais recebem ou estão disponíveis para transferir dados.

É usada apenas uma *Thread* para gerir estes canais de dados. Esta *Thread* lê as mensagens, deserializa-as e coloca-as na fila de comandos a processar. Para cada dispositivo conectado ao servidor é lhe associado uma chave que representa um canal e sempre que haja transferência de dados, o mecanismo deteta e recebe-os. Pelo mesmo canal é ainda possível transferir dados de volta para o controlador, sendo esta a forma utilizada para comunicar com os controladores sobre as informações do sistema.

Os dados recebidos vêm serializados através do mecanismo de *Proto Buffers*, tal como já foi referido na Secção 3.2.5.1.

Tempo de Vida - Cada controlador conectado, fica guardado no sistema e é lhe associado um tempo de vida. Os controladores são armazenados num mapa de controladores em que a sua chave é o seu identificador. Sempre que um controlador envia um novo comando, o tempo de vida associado a cada controlador é atualizado (ver Secção 2.3.1).

Para verificar se algum controlador não emitiu comandos no tempo de vida disponível, o sistema conta com um *Thread* dedicado a verificar esses tempos. Este percorre, periodicamente, todos os controladores do mapa e verifica o seu tempo de vida. Se algum não tiver emitido dados nesse tempo, este é removido do sistema, eliminando também as suas informações do histórico de controladores.

Devido a este mecanismo o impacto da mobilidade e *churn* é mínimo.

3.3.5.2 Processamento de Comandos

Depois de haver dados na fila de comandos, é possível iniciar o processamento. Para o processamento é usado o número de hardware *Threads* disponíveis na máquina, ou seja, será esse número menos os que estão a ser utilizados pelo sistema para receber comandos, para verificar o tempo de vida e para enviar. Para isso é usada uma *pool* de *Threads*. Na Figura 3.7, está esquematizado a forma de como o processamento de comandos é feita. Cada um dos *Workers*, apresentado na Figura 3.6, executado por um *Thread*, tem associado um identificador e é responsável por processar um segmento do fluxo de comandos. Para além do fluxo de comandos provenientes dos controladores, cada *Worker*, acede também ao segmento, do mesmo intervalo de tempo, dos comandos já processados por outros servidores do nível abaixo. Acede ainda, ao histórico de comandos, no caso de não estarem presentes no segmento retirado, todos os controladores e servidores conectados.

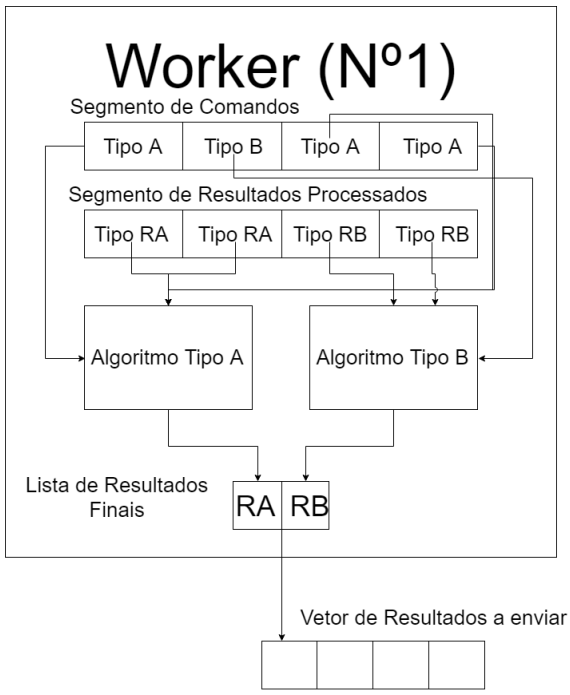

Figura 3.6: Esquema do funcionamento de um *Worker*

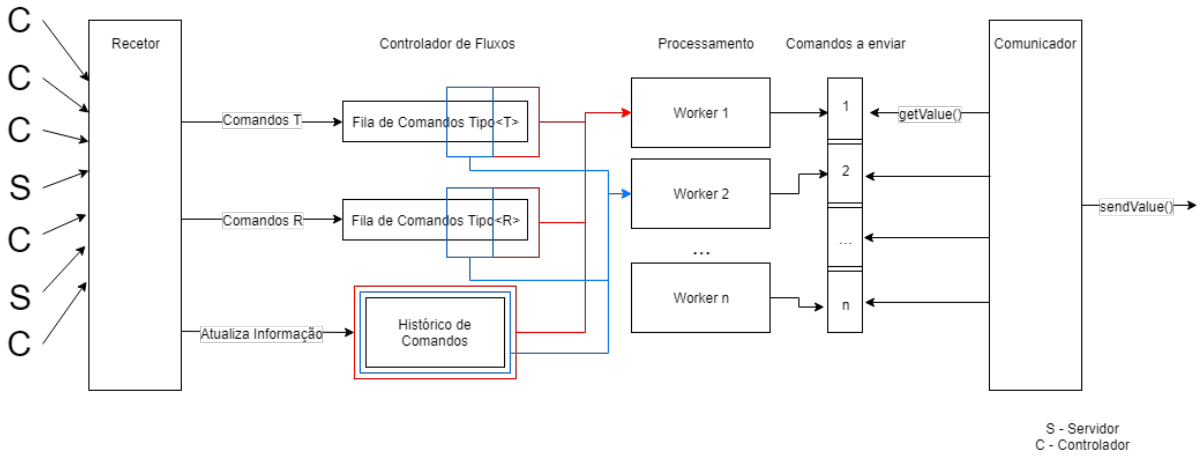

Figura 3.7: Esquema do Processamento por parte do Sistema

O resultado do processamento são os resultados provenientes dos algoritmos e são guardado num vetor de listas de comandos ordenado, por ordem do identificador dos vários *Workers*, até serem depois enviados.

Para controlar a concorrência no acesso às filas e ao histórico os métodos são sincronizados e para isso é utilizado a *keyword synchronized* disponibilizada pelo Java.

Os *Threads* que efetuam o processamento geram então objetos do tipo CommandList<Command><T> > e CommandList<Command<R> > de modo a aplicarem o algoritmo de agregação.

O objeto do tipo CommandList, extende uma *LinkedList* [45], que é fornecida pelo Java.

3.3.5.3 Envio da Informação

Por fim, o sistema conta ainda com uma *Thread* que envia os dados para o aplicação destino, ou para o servidor do nível acima. Este envia os dados através de uma conexão estabelecida, podendo ser utilizadas as tecnologias sem fios abordadas na Secção 2.2.1 ou até outra forma de comunicação, pois a aplicação destino pode estar a correr na mesma máquina do servidor principal. O envio de dados depende da implementação do método **void send(R result)** referido na Secção 3.3.2 por parte do programador.

Os dados a enviar estão armazenados no vetor de resultados que é preenchido pela *pool* de *Threads* descrito na Secção acima e exemplificada na Figura 3.7.

Contudo, existe a hipótese de não haverem dados nesse vetor. A*Thread* que envia os dados para a aplicação verifica periodicamente, consoante a taxa de frequência definida pelo programador, se tem dados a enviar e ao verificar as posições do vetor existe a possibilidade de uma destas estar vazia. Isto significa que o tempo de processamento por parte dos *Workers* não é rápido o suficiente para fornecer os dados ao nível acima com a frequência pedida.

Estes são os casos em que o sistema se tem de adaptar e enviar um pedido aos controladores e servidores conectados para que estes reduzam a cadência com que estes a emitir comandos. Por outro lado, depois desse valor ter sido adaptado e o sistema se estiver a comportar da forma correta, o número de comandos a emitir pode ser aumentado gradualmente, de modo a melhorar os resultados obtidos.

Exemplificando: a *Thread* que comunica os resultados, periodicamente acede a cada posição do vetor de resultados. Se uma destas posições estiver vazia, significa que o tempo de processamento por parte de um *worker* é demasiado elevado. De forma a libertar a carga em cada *worker*, o sistema envia um sinal aos controladores para que estes diminuam a sua frequência de envio. Depois disto o sistema dá um tempo aos *workers* de se adaptarem, ou seja, de computarem todos os dados que estavam em espera, esse tempo foi fixado em 3 segundos, podendo ser sendo facilmente alterado. Depois do sistema estar adaptado, é dado novamente atenção aos casos em que o vetor de resultados tenha posições vazias quando lhe é feito o acesso. Se for esse o caso é repetido o processo descrito anteriormente. Contudo, existe também o caso contrário, pois se o vetor de comandos contiver sempre todas as posições com valores, significa que o sistema tem espaço a melhorar. Sendo assim, definiu-se que se o comunicador percorrer o vetor de resultados cinco vezes com sucesso, é emitido aos controladores um sinal para que estes aumentem a sua frequência de envio. É importante realçar que o método que diminui a frequência de envio duplica o tempo de espera da *Thread* de envio presente nos controlador. Para reduzir a frequência diminuí-se esse valor uma vez e meia até o valor ser mínimo.

3.4 Dispositivo Móvel como Controlador e Servidor

Tal como abordado na Secção 2.3.2, existe a hipótese de um dispositivo móvel assumir o papel de servidor. Esta abordagem é possível, contudo existem alguns fatores a ter em conta como o hardware do controlador, a sua bateria, o número de controladores no sistema, a conexão, entre outros. A escolha de qual ou quais os dispositivos que podem ser servidores têm de ter em conta estes fatores, pois um dispositivo com poucas capacidades a nível de hardware ou com uma percentagem de bateria muito baixa, não é um bom candidato a ser utilizado como servidor. Posto isto, um dispositivo a utilizar será sempre um dos dispositivos com maiores capacidades a nível de processamento e também com bateria suficiente para aguentar os custos. Contudo a escolha de tal dispositivo não faz parte dos objetivos da tese, mas existem vários algoritmo como por exemplo, um algoritmo que escolha para servidores dispositivos otimizando a bateria, como é discutido no artigo [46].

O utilizador que manuseia esse controlador não irá sentir nenhuma diferença no sua experiência, pois o que é mostrado pela interface é exatamente o mesmo que é mostrado por um controlador normal. O que distingue os dois tipos de controlador é que um controlador que é servidor, tem as funções de um servidor normal, ou seja, estabelece-se num nível de hierarquia, recebe comandos de outros controladores e servidores, une os comandos recebidos e, por fim, envia-os para o nível acima na hierarquia. Contudo, existe uma diferença, pois não só tem de ter em conta os comandos que recebe através do recetor, como ainda tem de ter em conta os seus próprios comandos.

O código utilizado é o mesmo dos servidores estacionários, com algumas exceções. O número de controladores conectados é limitado, pois, não queremos sobrecarregar um controlador com comandos, isto porque, se compararmos o hardware de um dispositivo móvel com um servidor dedicado, é certo que o dispositivo terá capacidades inferiores.

Outra das diferenças é no processamento de comandos. O número de *Threads* a utilizar para o processamento será apenas de uma para evitar a sobre-utilização do equipamento.

Os comandos gerados localmente são igualmente colocados na fila de comandos a processar, contudo não existe a necessidade de fornecerem nenhuma mensagem para garantir que estão presentes. Essa informação é passada através de um método local.

Esta abordagem só será utilizada em casos em que o número de clientes seja muito elevado, o número de servidores estacionários seja baixo e o algoritmo a utilizar tenha tempos de processamento muito elevados. Noutros casos, não haverá necessidade, pois os servidores estacionários serão suficientes, aliviando a carga dos dispositivos móveis.

3.5 Sumário

Neste capítulo foi apresentado a *framework* desenvolvida e foi ainda apresentada a aplicação desenvolvida, ou seja, o protótipo do controlador utilizado. Foram apresentadas as

funcionalidades que a API fornece ao programador tal como os parâmetros obrigatórios que este tem de definir de modo a que o sistema tenha o comportamento correto.

Foram apresentadas todas as características do sistema desenvolvido assim como as formas de como são resolvidos problemas de escalabilidade, mobilidade e problemas relativos à carga na rede.

O próximo capítulo dedica-se à validação deste sistema desenvolvido, apresentando os resultados obtidos nos testes efetuados e é apresentado o caso de estudo utilizado em alguns desses testes.

Capítulo 4

Resultados Experimentais

Neste capítulo, pretende-se observar as características e o comportamento do sistema desenvolvido, em diferentes cenários. Para tal, foram realizados testes em dois ambientes de avaliação: ambiente real, utilizando dispositivos móveis, e ambiente simulado, utilizando uma implementação simulada da aplicação desenvolvida.

4.1 Métricas de avaliação

Para avaliar o comportamento, funcionamento e características do sistema desenvolvido, foram realizados testes em dois ambientes de avaliação, sendo eles: ambiente real e ambiente simulado.

Em ambiente real foram utilizados dispositivos móveis que permitiram verificar o comportamento do sistema e da aplicação desenvolvida. A comunicação destes dispositivos foi feita através de rede Wi-Fi. Nestes testes foi possível avaliar:

- o funcionamento do sistema;
- o impacto da granularidade;
- a taxa de entrada e saída de comandos de um controlador que é servidor;
- o consumo energético.

Os testes em ambiente real foram úteis de modo a comprovar que é possível a utilização deste sistema numa aplicação real, como o caso do jogo do *Pong*, descrito na Secção 4.2. Sendo assim, pode ser utilizada em outras aplicações desde que seja necessária a cooperação entre estes.

Em ambiente simulado, foi utilizada uma camada de rede que permite a comunicação entre os vários nós e os servidores. Desta forma, foi possível simulador um número

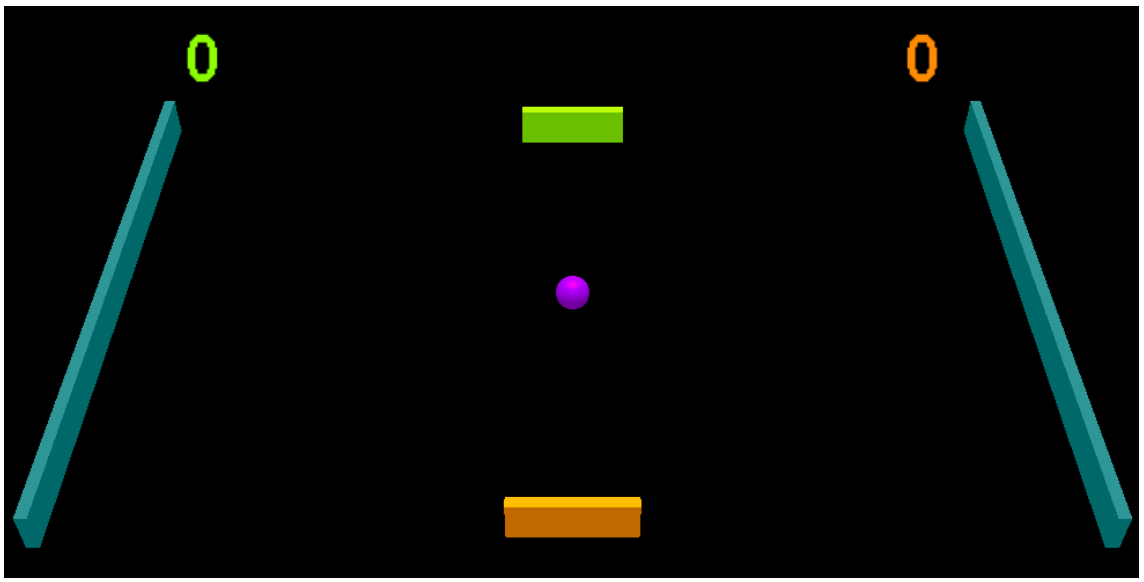

Figura 4.1: Jogo do Pong

bastante elevado de nós, que se comportam como dispositivos comuns. Foi necessário fazer adaptações, para substituir os sensores, ou seja, em vez da utilização de sensores, os dispositivos simulados, geram um número aleatório com o valor emitido pelo sensor, com um dado intervalo de tempo, também aleatório, de modo a simular os comandos. Este ambiente simulado foi desenvolvido em Java e foram utilizadas várias máquinas, de modo a simular uma grande quantidade de controladores.

Com este ambiente foi possível avaliar:

- a escalabilidade e o seu impacto;
- a taxa de entrada e saída de comandos em cada nó;

4.2 Caso de Estudo

Como caso de estudo, foi utilizado um jogo, que se assemelha ao jogo do *Pong*, desenvolvido pela empresa *Atari* [10], como está descrito na Secção 1.2. O jogo utilizado está apresentado na Figura 4.1

O jogo que utilizado foi desenvolvido pelo Departamento de Informática da Faculdade Nova de Lisboa, em que cada barra vertical é controlada através de um sistema de votos. É utilizada uma câmara que deteta a quantidade de votos. Esses votos são contabilizados através do número de cartões que a câmara deteta. Se houver mais cartões do lado esquerdo a barra move-se para esse lado, caso contrário, move-se para o outro.

Neste caso de estudo, adaptámos o jogo que foi desenvolvido na faculdade. Em vez de se utilizar uma câmara para contar os votos, utilizou-se as informações recebidas dos vários dispositivos móveis. Para isso, foi utilizada a aplicação descrita na Secção 3.2.

Listagem 4.1: Objeto Algoritmo

```
Algorithm algorithm = new Algorithm() {
  @Override
  public Pair<Integer, Integer> process(CommandList cList,
  CommandList processedCommands, int weight) {
    int left = 0;
    int right = 0;
    int mid = 0;
    int w = weight;
    cList.removeDuplicates();

    for(Command<Integer> c : cList){
      if (c.getValue() < 70 && c.getValue() > 0) {
        right++;
      } else if (c.getValue() > 70 && c.getValue() < 110) {
        mid++;
      } else if (c.getValue() > 110) {
        left++;
      }
    }
    for(Command<Integer> c : processedCommands){
      w += c.getWeight();
      if( c.getValue() ==1)
        right+=c.getWeight();
      else if(c.getValue())==0)
        mid+=c.getWeight();
      else
        left+=c.getWeight();
    }

    int result;
    if(right == left) result = 0;
    else if(right >= left+mid) result = 1;
    else if(left >= right + mid) result = -1;
    else result = 0;

    return new Pair(result, w);
  }
};
```

A aplicação comunica com o sistema através da tecnologia Wi-Fi. Os utilizadores inclinam o seu controlador para um dos lados, de modo a votar na direção que pretendem que a barra se desloque. Caso não pretendam que a barra mude de posição, devem manter os seus dispositivos imóveis, em posição horizontal.

Em vez da utilização da câmara, são utilizados servidores, que processam a informação e a enviam para o jogo.

O algoritmo de decisão é um algoritmo de consenso, ou seja, se a quantidade de dispositivos que escolham, por exemplo, que a barra se desloque para a esquerda for maior que a quantidade que pretende que ela que imóvel mais o número de dispositivos que pretende que a barra se desloque para a direita, então a barra move-se nessa direção.

Na Listagem 4.1 é apresentado o algoritmo utilizado.

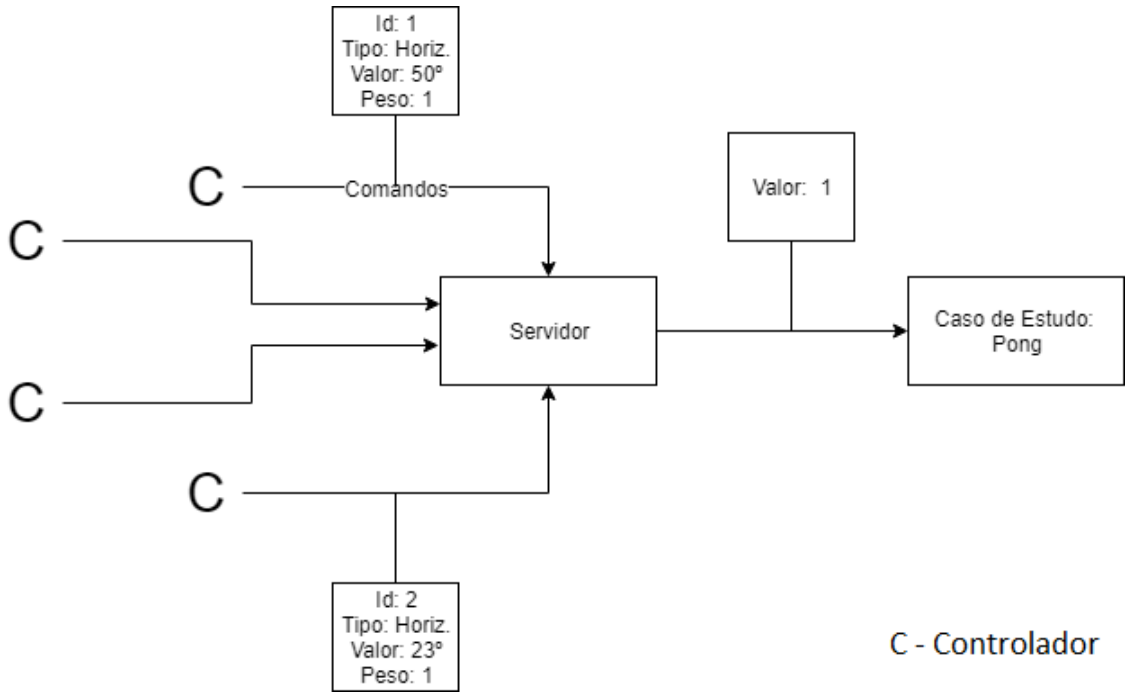

Figura 4.2: Exemplo da Informação Transferida

A frequência pedida pela aplicação é de um comando a cada 200 milissegundos, por isso, a cada 200 milissegundos verifica-se se há comandos a enviar, caso haja esses comandos são enviados para a aplicação, caso contrário, o servidor adapta-se.

Na Figura 4.2, é apresentado um esquema que ilustra a informação passada na rede, isto é, os dispositivos emitem comandos, com valores em graus que são disponibilizados pelos seus sensores.

O servidor processa esses comandos e converte o resultado no tipo pedido pela aplicação decisão, neste caso um número inteiro entre -1 e 1, e envia-o.

4.2.1 Aplicação Desenvolvida

Foi desenvolvido um protótipo de um controlador, de forma a ser possível a validação do sistema. Para além disso, esse protótipo clarifica o comportamento de um controlador. A aplicação desenvolvida destina-se a ser aplicada no jogo do *Pong*, descrito na Secção 4.2.

Inicialmente a aplicação apresenta apenas um ecrã com apenas uma opção possível "Press to Start". Ao pressionar o ecrã, é iniciada a busca por servidores e conecta a aplicação a um destes, tal como referido na Secção 3.2.2. Este ecrã é apresentado na Figura 4.3.

Depois da conexão ao servidor estar estabelecida este envia informações ao controlador. Neste caso o servidor fornece as informações sobre o nome do jogo, o número de equipas e as equipas disponíveis. Ao utilizador do dispositivo é mostrada essa informação, dando a opção de este escolher a sua equipa desejada, exemplificada na Figura 4.4.

Depois do utilizador escolher a sua equipa, começa a enviar comandos para o sistema. O método de envio de comandos na aplicação desenvolvida é através da inclinação do

Figura 4.3: Ecrã Inicial da Aplicação

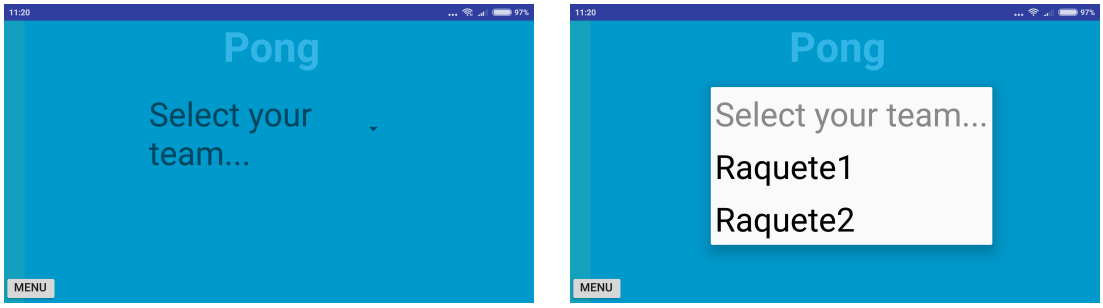

Figura 4.4: Escolha da Equipa

dispositivo. É mostrado ao utilizador se este está ou não a enviar comandos, através de barras verticais colocadas nas extremidades horizontais do ecrã, como está representado na Figura 4.5. As barras aparecem e desaparecem consoante a inclinação do dispositivo móvel e quanto maior a intensidade da sua cor, maior a inclinação do dispositivo. Caso nenhuma das barras esteja colorida, significa que o utilizador não pretende que a sua personagem no jogo se mova, isto é, controlador encontra-se na posição horizontal.

4.3 Setup Inicial

Nos testes que são apresentados nas Secções 4.4 e 4.5 foram utilizados três algoritmos de processamento. Nos testes em ambiente real (Secção 4.4) foram utilizados dois desses algoritmos. O primeiro é o algoritmo já descrito na Listagem 4.1, em que funciona como um sistema de votos. O tempo de processamento deste algoritmo é bastante baixo e por essa razão, para alguns dos testes que são avaliados na Secção 4.4 utilizou-se também um algoritmo de tempo de processamento médio por comando fixo, sendo este tempo de 25ms.

Para os testes em ambiente simulado (Secção 4.5), foram utilizados os dois algoritmos já descritos em cima e para além destes, foi ainda utilizado um terceiro algoritmo de tempo de processamento médio de 50ms por comando.

Em todos os testes efetuados, a frequência de saída de comandos desejada é de pelo menos 4 comandos por segundo.

De forma a retirar-se resultados no contexto em que o número de controladores ligados

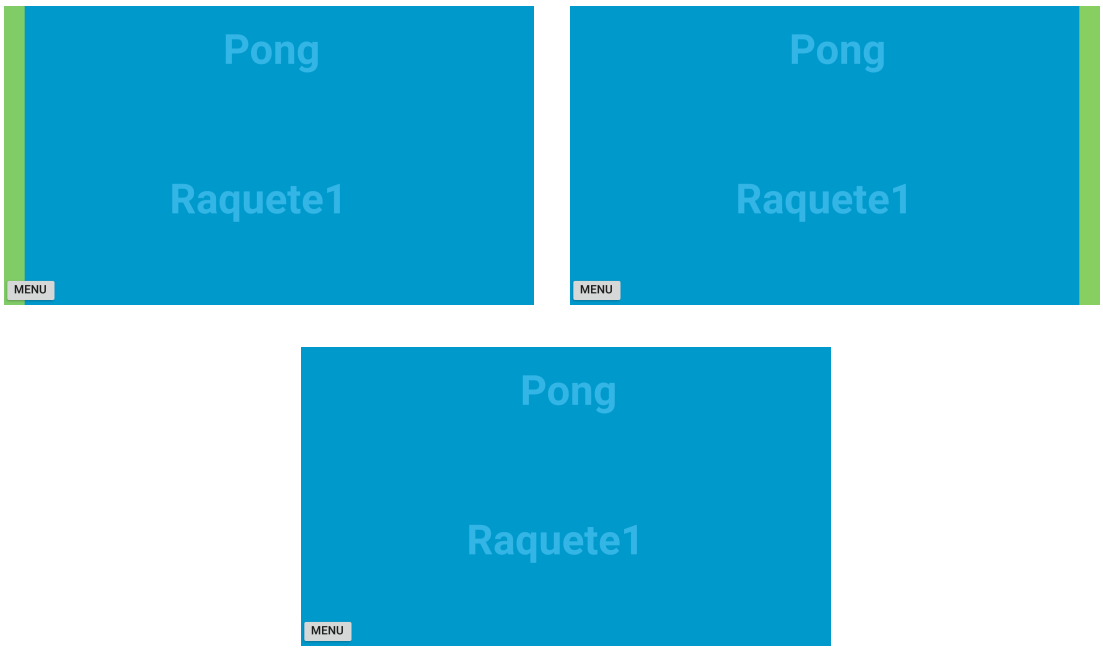

Figura 4.5: Movimento do dispositivo de forma a enviar comandos

ao sistema é elevado, utilizou-se um simulador de forma a criar controladores virtuais de modo a testar o sistema implementado. Estes controladores emitem um comando de tempo a tempo, sendo este tempo aleatório, tendo um limite máximo de 200ms.

As máquinas usadas como servidores têm como características principais um processador Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz e 64Gb de memória RAM. Nos testes executados foram usadas 19 *Threads* de processamento. A latência nestes casos foi desprezada.

Os controladores conectam-se através de *web sockets*

4.4 Testes Ambiente Real

4.4.1 Funcionamento do Sistema

Para verificar o bom funcionamento do sistema foram efetuados testes em que vários utilizadores, jogaram uns contra os outros, sendo que o resultado final foi o esperado, verificando-se o bom funcionamento do jogo descrito no caso de estudo. Devido à falta de recursos, apenas foram feitos testes com um número de reduzido de utilizadores.

Neste teste a conexão foi feita através de Wi-Fi.

4.4.2 Tempo de um Comando

Foi feito um pequeno teste em para verificar quanto tempo demora um comando desde que saí do controlador até chegar a aplicação. O resultado está apresentado na figura 4.6.

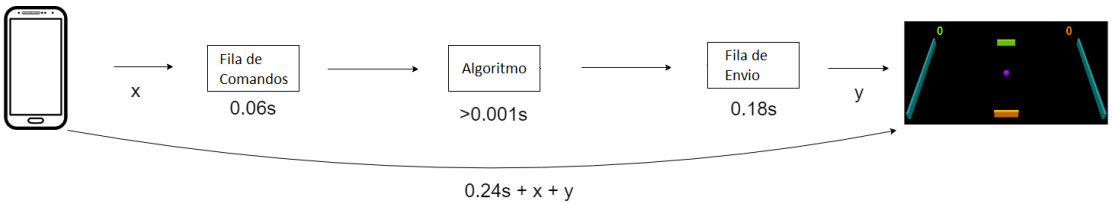

Figura 4.6: Tempo de um comando no sistema

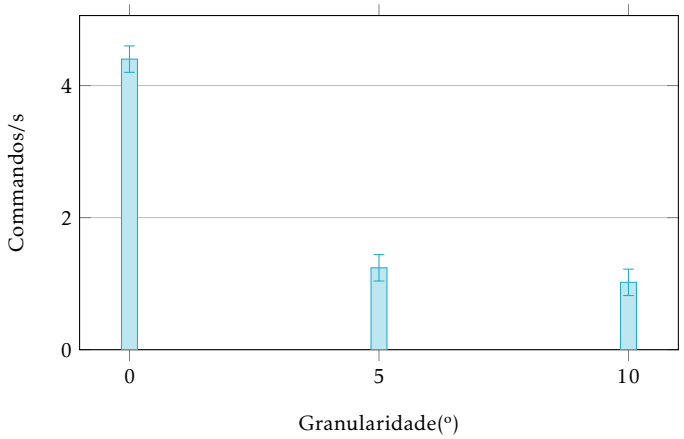

Figura 4.7: Número de comandos enviados consoante a granulosidade

Em média, um comando demora 0.24s no servidor. Para sabermos ao certo quanto tempo demora desde que o comando sai do controlador até chegar a aplicação, depende de fatores externos, como a velocidade na rede.

4.4.3 Granularidade

A granularidade consiste na diferença significativa de comandos consecutivos de modo a evitar envio de comandos semelhantes. Para verificar o seu impacto e a quantidade de comandos enviados, foram efetuados testes com um número reduzido de dispositivos para que a importância de um cliente fosse máxima. Sendo assim, variou-se a granularidade em graus e um controlador só enviou o seu comando para o servidor se a diferença entre dois comandos consecutivos fosse maior que a granularidade definida. Foram efetuados testes com granularidade de 0, 5 e 10º e o número de comandos emitidos pelo dispositivo variam consideravelmente como está representado na Figura 4.7

Os testes decorreram durante, aproximadamente, 5 minutos e a diferença entre granularidade 0 e 5º é significativamente grande, o que significa um grande número de mensagem enviadas por parte de dispositivos com 0 de granularidade. Isto tem impacto direto na jogabilidade, pois com 0º, um grande número de comandos semelhantes é captado pelo servidor, verificando-se algum atraso no movimento da barra do jogo, pois se os comandos emitidos são muito semelhantes é como se o jogo recebesse comandos em duplicado, demorando mais tempo a processar, havendo assim atrasos no movimento da raquete.

Com granularidade de 5º, o jogo decorre dentro da normalidade, sendo esta a granularidade desejada neste caso de estudo. É desta forma que o jogo se comporta melhor, dando aos utilizadores uma melhor experiência.

Com o aumento da granularidade, o jogo comporta-se de maneira semelhante, contudo piora a experiência do utilizador, pois os movimentos que este tem de fazer com o seu dispositivo são muito mais bruscos (os movimentos mais suaves dão a sensação de não ter impacto), tornando a experiência desconfortável do ponto de vista de um jogador.

Assim sendo, neste caso de estudo o que melhor se adapta é uma granularidade de 5º, mas não significa que noutras aplicações se comporte da mesma forma. A granularidade é inversamente proporcional ao número de comandos a emitir, por isso aplicações que necessitem de um maior número de comandos ou comandos mais refinados, necessitam também de granularidade mais baixa. Da mesma forma de aplicações mais simples, em que não existe necessidade de um grande número de comandos, deverá utilizar-se uma granularidade mais elevada.

4.4.4 Processamento de comandos por parte dos dispositivos e consumo energético

Para verificar o comportamento de um dispositivo móvel a nível de processamento, não foi utilizado o algoritmo utilizado no caso de estudo, mas sim o algoritmo com maior tempo de computação, que nos permite tirar maiores conclusões sobre o comportamento dos dispositivos.

Para este teste, utilizou-se como servidor um telemóvel Xiaomi Redmi 4, que conta com um CPU Octa-core 2.0 GHz Cortex-A53 e ainda com 4GB de memória RAM. Foi utilizado ainda um servidor estacionário, para o qual foram enviados os comandos processados pelo dispositivo. Os controladores conectados ao telemóvel foram simulados em software, no computador, e a conexão foi feita através de Wi-Fi.

Os resultados obtidos foram que o controlador consegue emitir uma boa frequência de comandos, mesmo com um número elevados de dispositivos conectados. Na pior das hipóteses testadas, com 25 controladores conectados mais 1, que é o próprio dispositivo, a taxa de saída média manteve-se acima dos 4 comandos/s.

Assim, temos garantias de que se houver necessidade de aliviar a carga dos servidores, dispositivos como o que foi utilizado ou outros com hardware semelhante, conseguirão fazer esse processamento de forma eficaz.

O problema que surge é apenas relacionado com o seu consumo energético, e neste teste também foram retirados os valores de consumo, como estão é apresentado na Figura 4.9.

As medições de consumo de bateria foram efetuadas automaticamente, através de um módulo que utiliza a classe BatteryManager fornecida pelo Android.

O caso com 0 nós conectados, serve de comparação entre um cliente normal da aplicação e um servidor.

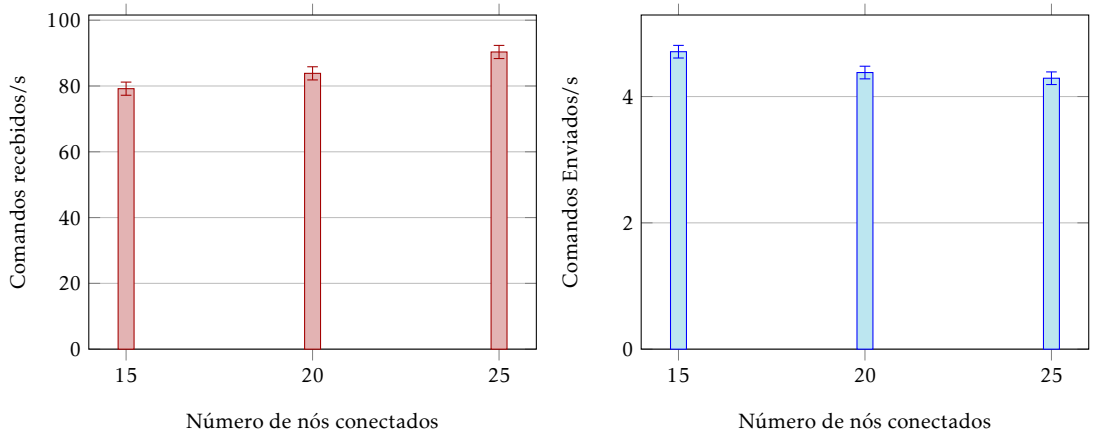

Figura 4.8: Taxa de Entrada e Saída processamento por parte de dispositivos que são servidores.

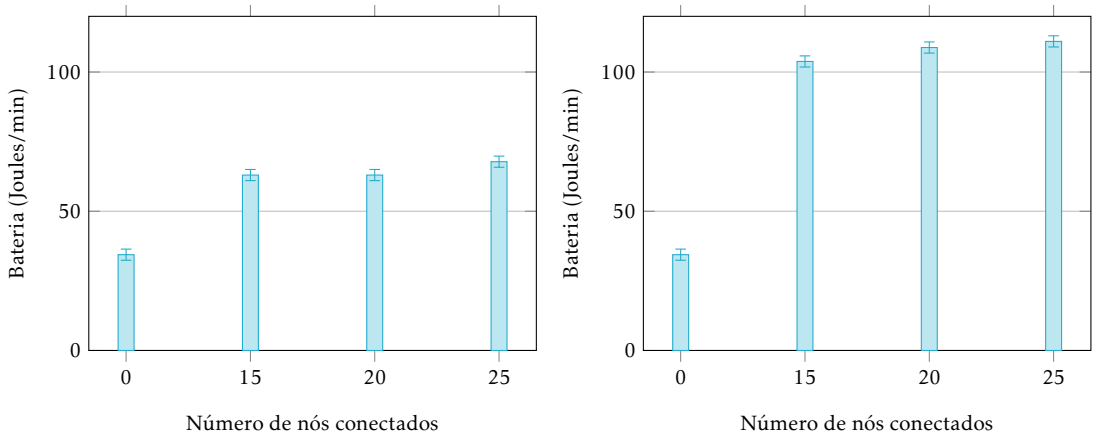

Figura 4.9: Consumos de energia de um dispositivo que é utilizado como servidor

No gráfico da esquerda, da Figura 4.9, foi utilizado o mesmo algoritmo que foi utilizado para retirar resultados sobre o processamento.

O telemóvel utilizado conta com uma bateria de 4100mAh. Dando uma perspetiva em joule, que é a unidade de medida utilizada nos testes, 1% da bateria de um Xiaomi Redmi Note 4 equivale a aproximadamente 590 joule. Nos testes efetuados, um dispositivo que suporta 20 nós, gasta aproximadamente entre 60 e 65 joule por minuto, ou seja, em 10 minutos de utilização constante da aplicação e ainda servindo de servidor do sistema, estima-se que gaste aproximadamente, pouco mais de 1% da bateria do dispositivo.

Ao fazer o mesmo teste, mas utilizando o algoritmo do jogo do *Pong* (caso de estudo) e com granularidade de 5º, como o algoritmo é demasiado simples e leve, o tempo médio de processamento é desprezável, sendo assim, o único valor de interesse é o seu consumo energético e esse está representado na à direita na Figura 4.9.

Com este algoritmo o consumo energético é um pouco maior, isto deve-se ao facto de correr esse código mais vezes, pois o custo energético da discretização e da verificação do histórico é superior ao custo do processamento desses mesmos comandos. Sendo assim, no

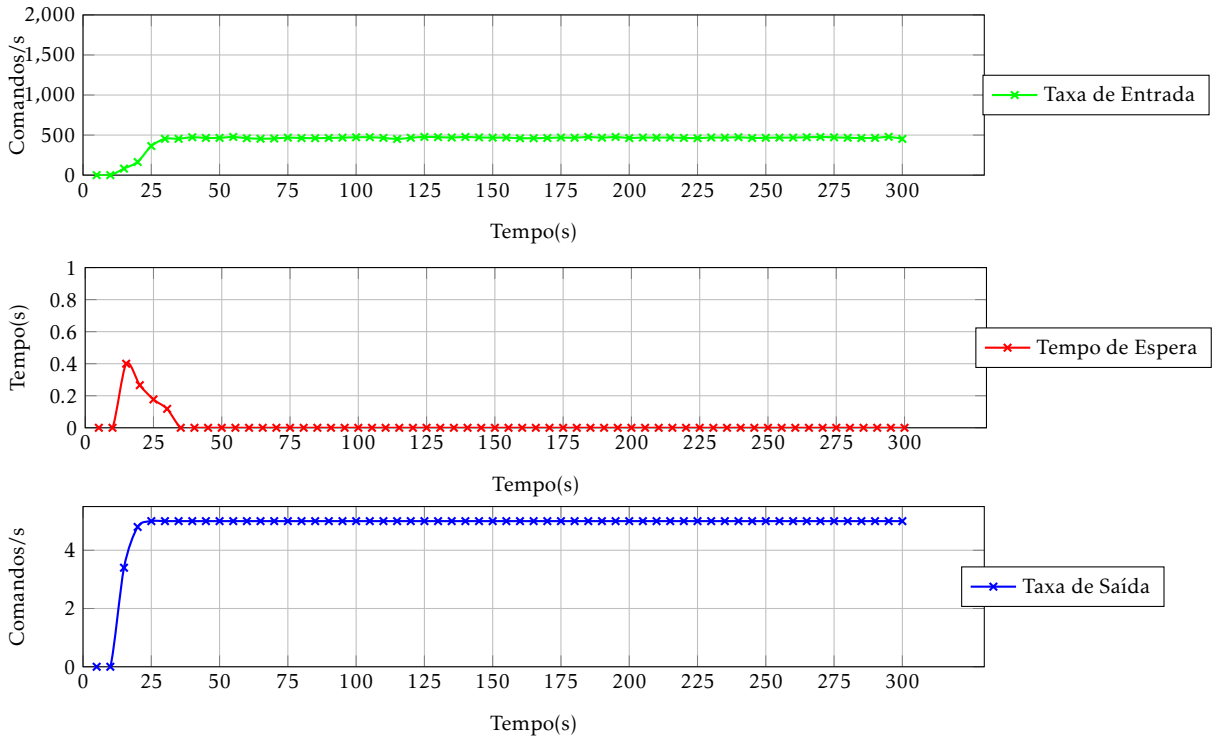

Figura 4.10: Algoritmo de consenso do caso de estudo, 1 servidor, 50 clientes

caso do algoritmo de consenso usado no *Pong*, o consumo energético para um dispositivo que suporta 20 nós é de aproximadamente 110 Joule/min. Isto é, em 10 minutos de utilização estima-se que gaste aproximadamente 1,9% da bateria do telemóvel utilizado.

4.5 Testes Simulador

4.5.1 Performance de um Servidor

Inicialmente utilizou-se o algoritmo de consenso utilizado no caso de estudo, foi utilizado um servidor e efetuou-se dois testes. O primeiro, conta com 50 controladores conectados e foram retirados valores quanto à sua taxa de entrada, taxa de saída e tempo de espera. O tempo de espera é o tempo que um controlador adia o seu envio de comandos de forma reduzir a frequência de envio, porque o *bandwidth* não parece ser o factor de *bottleneck*.

Os resultados obtidos neste teste estão apresentados na Figura 4.10.

Como o tempo de processamento é mínimo, não houve necessidade de adaptação do tempo de espera.

Em seguida, testou-se o mesmo algoritmo mas com um número considerável de controladores: 200. O resultado no teste encontra-se na Figura4.11.

Comparando este caso com o anterior, houve um aumento significativo na taxa de entrada, em que a taxa média de entrada com 50 controladores é de aproximadamente

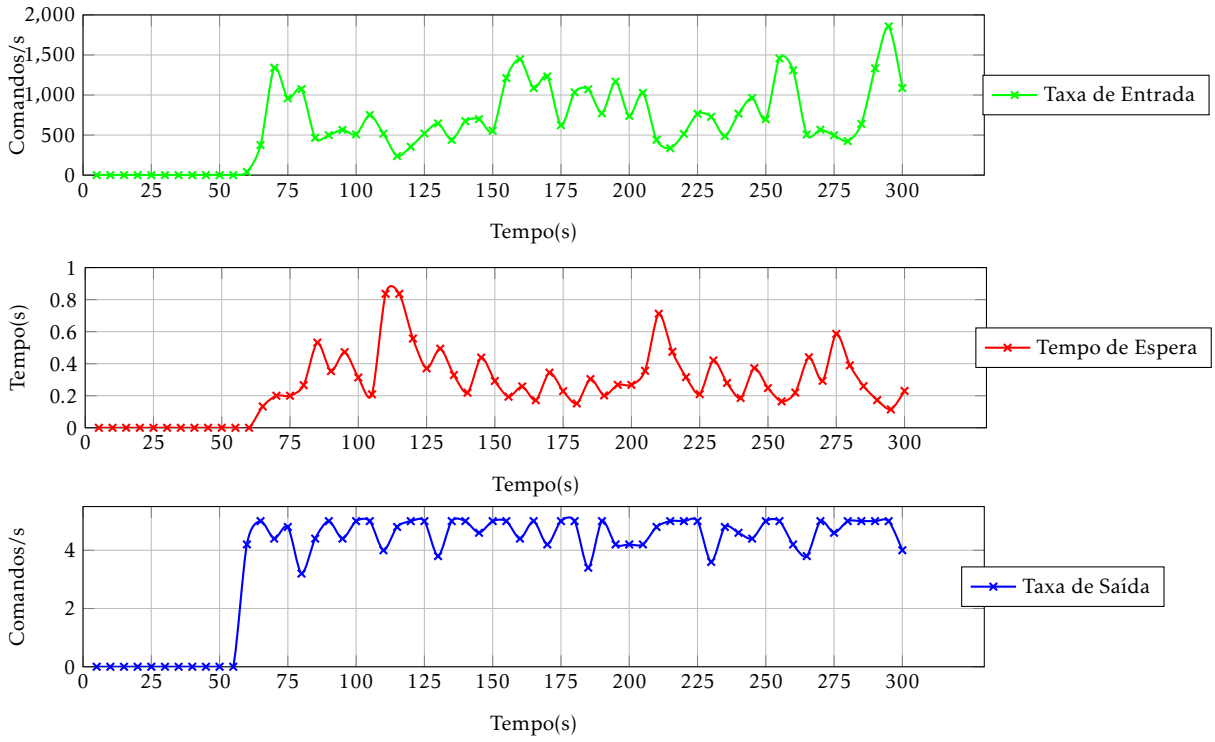

Figura 4.11: Algoritmo de consenso do caso de estudo, 1 servidor, 200 clientes

420 enquanto com 200 controladores é de 790. A taxa de saída no caso com menos controladores é mais regular, enquanto com 200 houve mais oscilações. Contudo o número de comandos enviados manteve-se sempre muito perto da frequência desejada.

A grande diferença entre estes dois testes está na adaptação do servidor. No segundo caso, ao contrário do primeiro, existiu adaptação, o que significa que houveram períodos em que o número de comandos para processar foi muito elevado, gerando muito tempo de processamento, fazendo com que o tempo de espera aumentasse. Contudo, este é o resultado esperado nestes casos, ou seja, se a carga no servidor for demasiado elevada este tem de comunicar aos controladores de forma a que a carga na rede baixe, diminuindo os tempos de processamento.

De seguida, foi utilizado o algoritmo de tempo de processamento médio por comando de 25 milissegundos, e foi utilizado apenas um servidor em que a variável foi o número de controladores conectados, sendo que esse número varia entre 50 e 200 controladores. Foram retirados valores de taxa de entrada e saída de comandos e ainda valores quanto à variação do tempo de adaptação por parte dos servidores, tal como nos casos anteriores.

Na Figura 4.12 é apresentado o caso mais simples desta secção. Estão conectados 50 controladores a um único servidor. Descartando os segundos iniciais da conexão de controladores ao servidor (primeiros 25 segundos), obteve-se uma taxa média de entrada de 259 comandos por segundo e uma taxa média de saída muito perto da desejada. Devido à simplicidade do teste e boa capacidade de processamento por parte do servidor, as

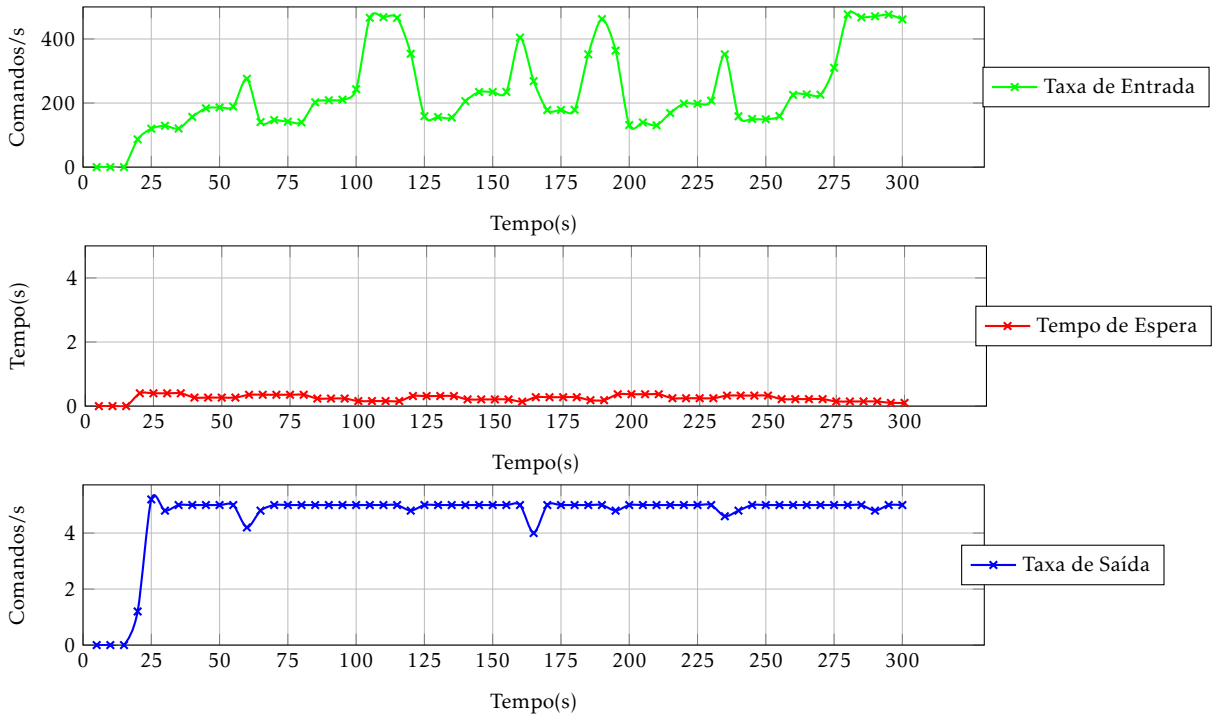

Figura 4.12: Algoritmo tempo médio de processamento de 25ms, 1 servidor, 50 clientes

adaptações existentes são muito baixas, sendo que o tempo de espera máximo nunca ultrapassou os 0.4 segundos

De seguida, aumentou-se o número de controladores ligados para 100 e esse teste está apresentado na Figura 4.13.

O número de comandos processados diminuiu ligeiramente, comparando com o caso anterior, 179 comandos por segundo, isto deve-se ao facto do tempo de espera ter aumentado. Devido a uma maior carga na rede, existiu uma maior necessidade de haver adaptações por parte do servidor. Se verificarmos o segundo 225 na Figura 4.13, conseguimos observar que houve um grande número de comandos a entrar no sistema e por essa razão o tempo de processamento aumentou demasiado fazendo com que a taxa de saída diminuí-se bastante, mas o sistema comportou-se de forma correta, aumentando o tempo de espera, reduzindo a taxa de entrada, conseguindo assim estabilizar a sua taxa de saída.

O caso seguinte em que são conectados 150 controladores ao servidor, representado pela Figura 4.14, tem um comportamento semelhante ao anterior, tendo uma taxa média de entrada um pouco menor, 126 comandos por segundo. O tempo de espera aumenta consoante o número de dispositivos ligados o que faz com que a taxa de entrada se mantenha baixa apesar do maior número de controladores. A adaptação neste caso comportou-se mais uma vez da maneira desejada, tendo em conta que a taxa de saída manteve-se com uma média muito perto da desejada.

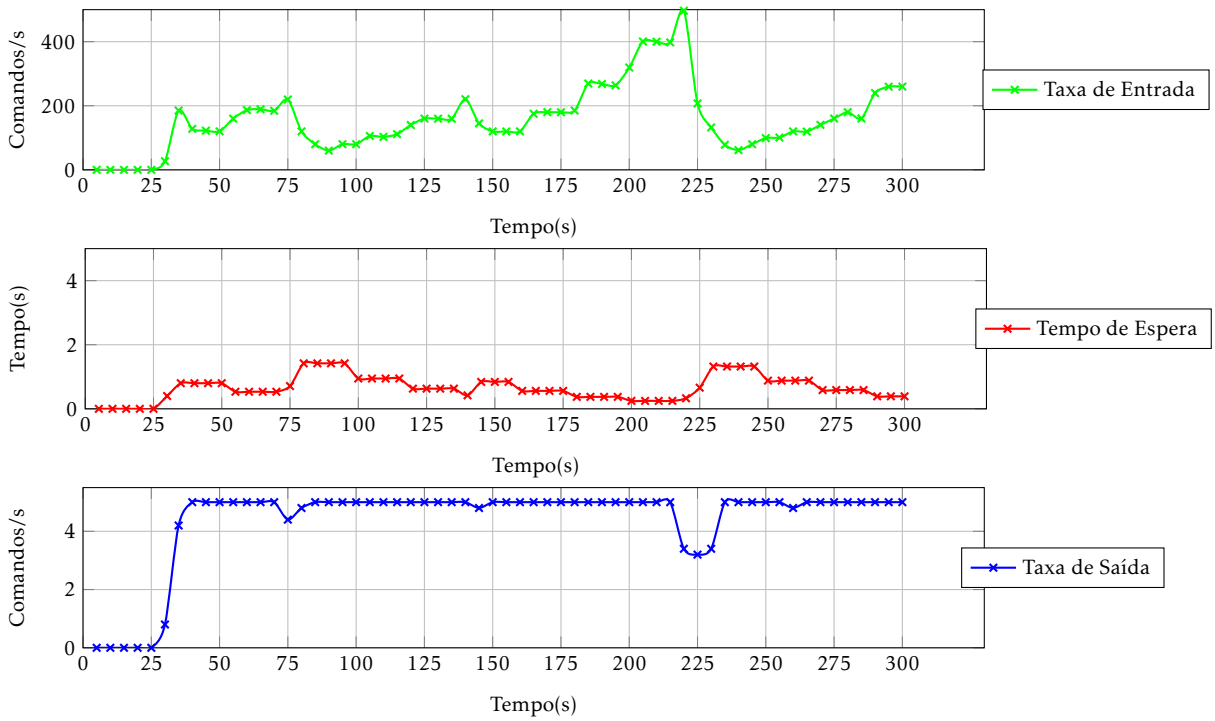

Figura 4.13: Algoritmo tempo médio de processamento de 25ms, 1 servidor, 100 clientes

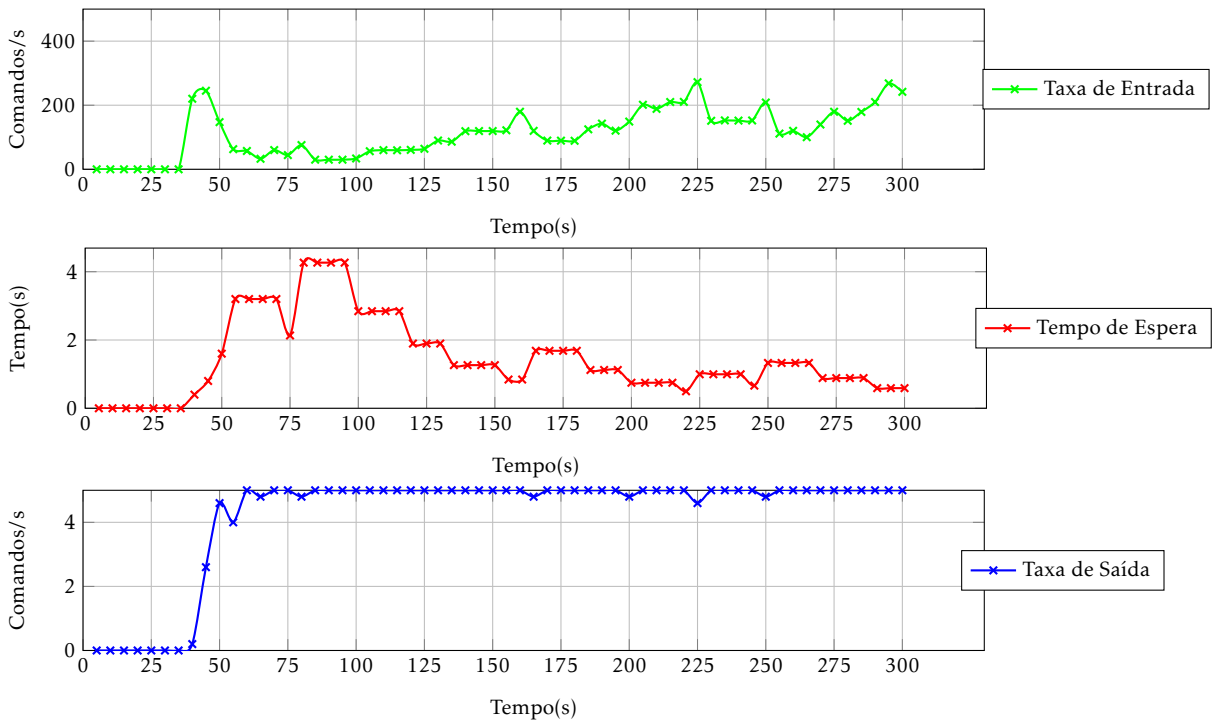

Figura 4.14: Algoritmo tempo médio de processamento de 25ms, 1 servidor, 150 clientes

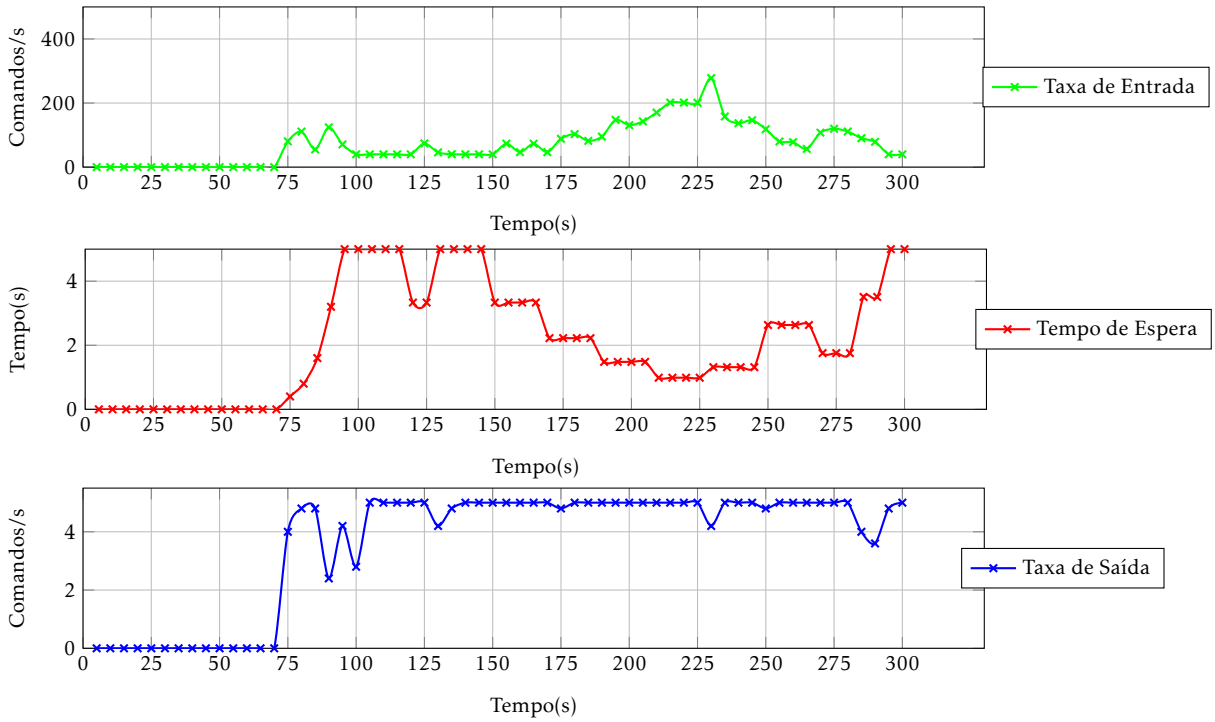

Figura 4.15: Algoritmo tempo médio de processamento de 25ms, 1 servidor, 200 clientes

O último caso, 200 controladores, teve um comportamento semelhante dos dois anteriores. Está representado pela Figura 4.15. Neste caso, a taxa de entrada ronda os 94 comandos. Um maior número de controladores conectados, não tem necessariamente de ter uma taxa de entrada maior. Como o número de controladores conectados é muito grande, o servidor tem a necessidade de reduzir a carga na rede, e por isso, o tempo de espera que o servidor impõe aos controladores nunca é elevado, atingido valores de 5 segundos, fazendo com que o número de comandos recebidos seja relativamente baixo. Desta forma o servidor consegue ir emitindo comandos com a frequência desejada.

De seguida, para testar a performance do servidor desenvolvido, aumentou-se o tempo de processamento por comando, ou seja, alterou-se o algoritmo de forma a consumir mais tempo a cada *Thread* de processamento, sendo que nos casos anteriores o tempo médio de processamento foi de 25 milissegundos e nos seguintes é de 50 milissegundos. Os resultados são apresentados nas Figuras 4.16, 4.17, 4.18 e 4.19.

No teste com apenas 50 clientes, representado pela Figura 4.16, nota-se diferenças consideráveis se compararmos com a Figura 4.12. Neste teste existem oscilações do tempo de espera apesar do número de utilizadores ser reduzido. A taxa de entrada de comandos no sistema foi de 111 comandos por segundo, enquanto a taxa de saída na maior parte do tempo foi a desejada (5 comandos por segundo) contudo houve momentos em que esse número foi mais baixo.

Aumentando o número de clientes no sistema para 100, como está apresentado na Figura 4.17, os resultados em termos de oscilações são muito maiores ao caso anterior, ou

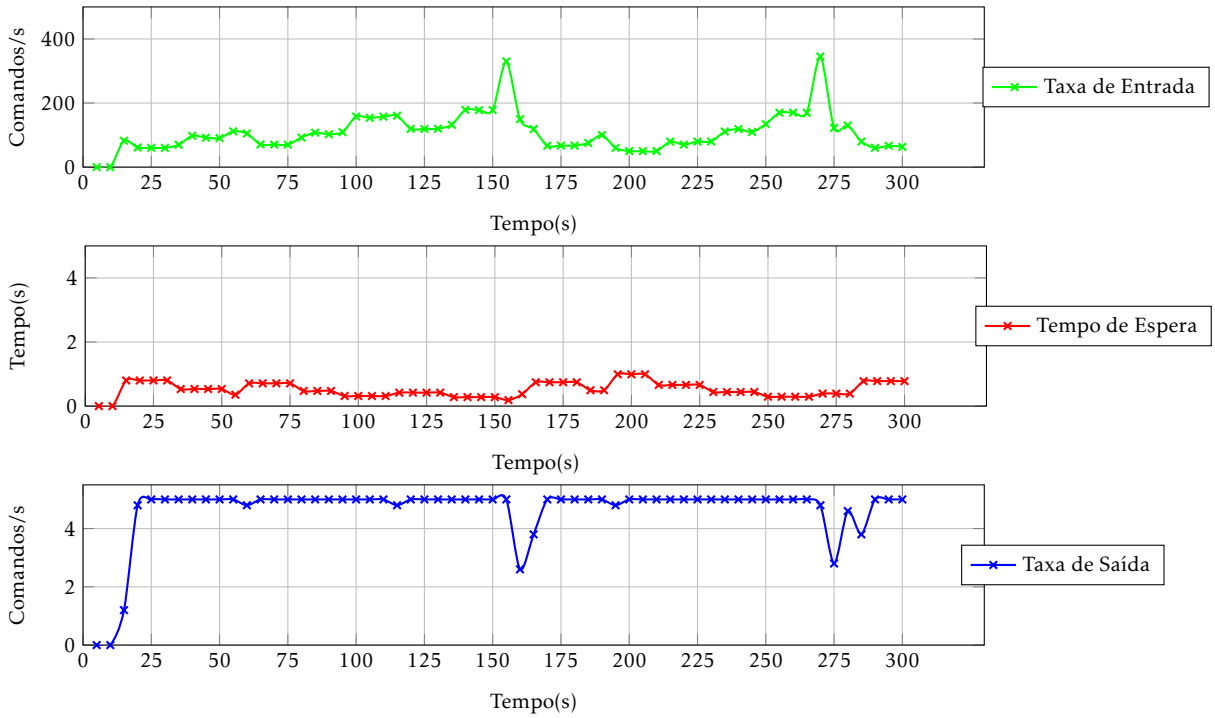

Figura 4.16: Algoritmo tempo médio de processamento de 50ms, 1 servidor, 50 clientes

seja, no caso com 50 clientes ligados o tempo de espera máximo nunca ultrapassa os 2 segundos enquanto que neste caso chega a valores de 5 segundos. Quanto maior a carga no servidor maior o tempo de espera que este impõe aos seus controladores conectados.

O teste com 150 clientes, representado pela Figura 4.18, tem comportamento semelhante ao teste anterior, a nível de oscilações.

No último e pior dos casos, fez-se ainda o mesmo teste mas com um número bastante elevado de controladores, 200 e o resultado é apresentado na Figura 4.19.

Com estes testes podemos retirar que o sistema dá prioridade à taxa de saída. Desta forma a aplicação destino terá sempre um número muito aproximado de comandos que pretende. Contudo, de forma a manter a taxa de saída desejada o sistema tem necessidade de reduzir a sua carga, informando os seus controladores conectados que baixem a sua frequência, e por isso o número de comandos processados é baixo. De forma a aumentar a taxa de entrada, ou seja, para que seja dada maior importância aos controladores, aumentou-se o número de servidores disponíveis e próxima secção apresenta os resultados obtidos com um maior número de servidores.

4.5.2 Escalabilidade

Para testar o impacto da escalabilidade no sistema, utilizou-se o algoritmo utilizado nos exemplos iniciais da secção anterior (25 milissegundos de processamento médio) e aumentou-se o número de servidores disponíveis para 2, distribuindo-se a carga de

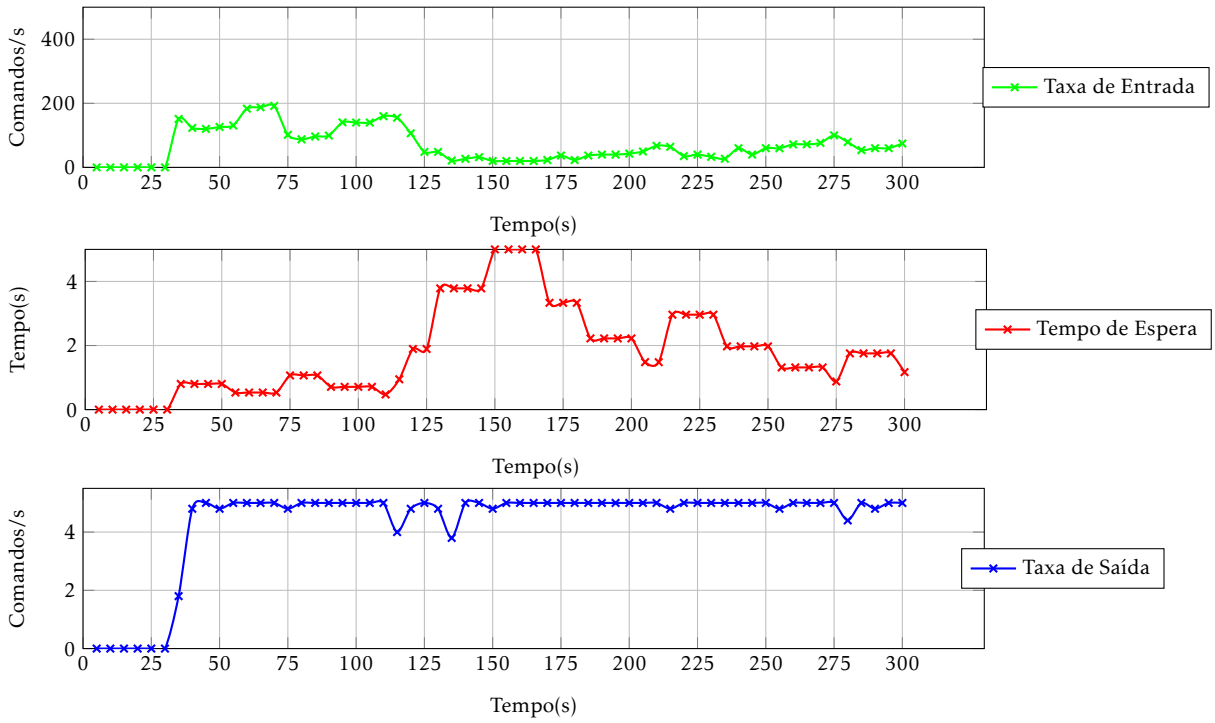

Figura 4.17: Algoritmo tempo médio de processamento de 50ms, 1 servidor, 100 clientes

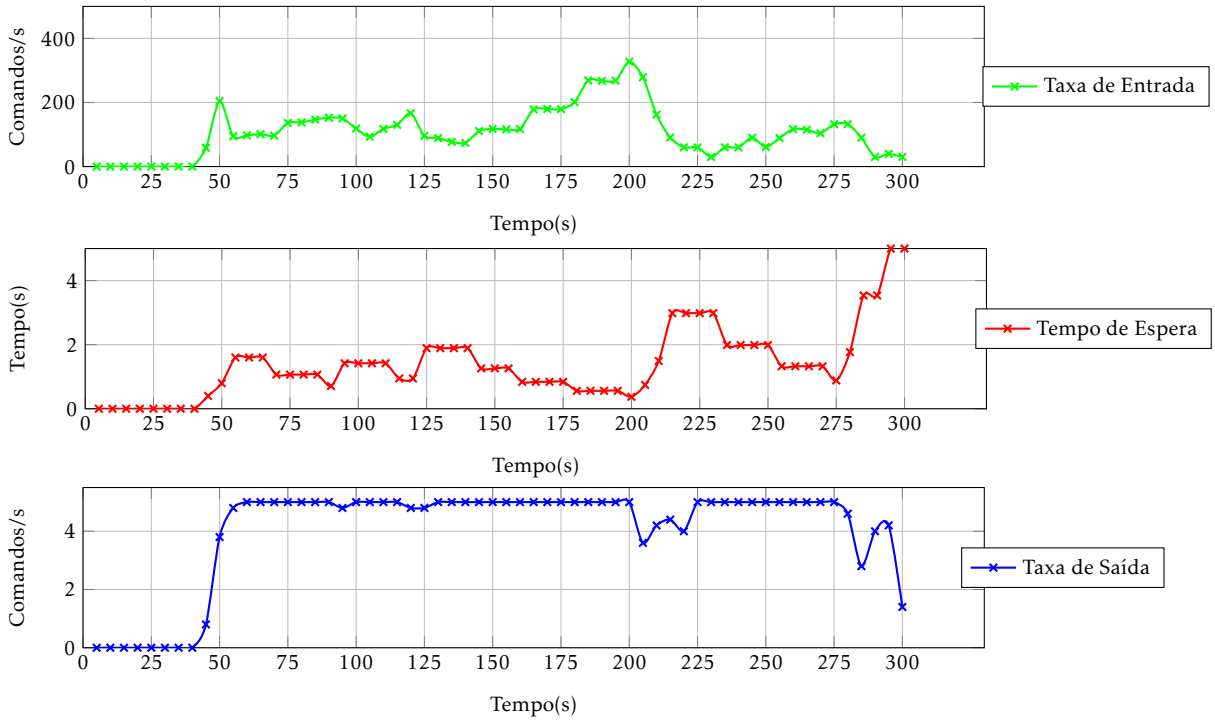

Figura 4.18: Algoritmo tempo médio de processamento de 50ms, 1 servidor, 150 clientes

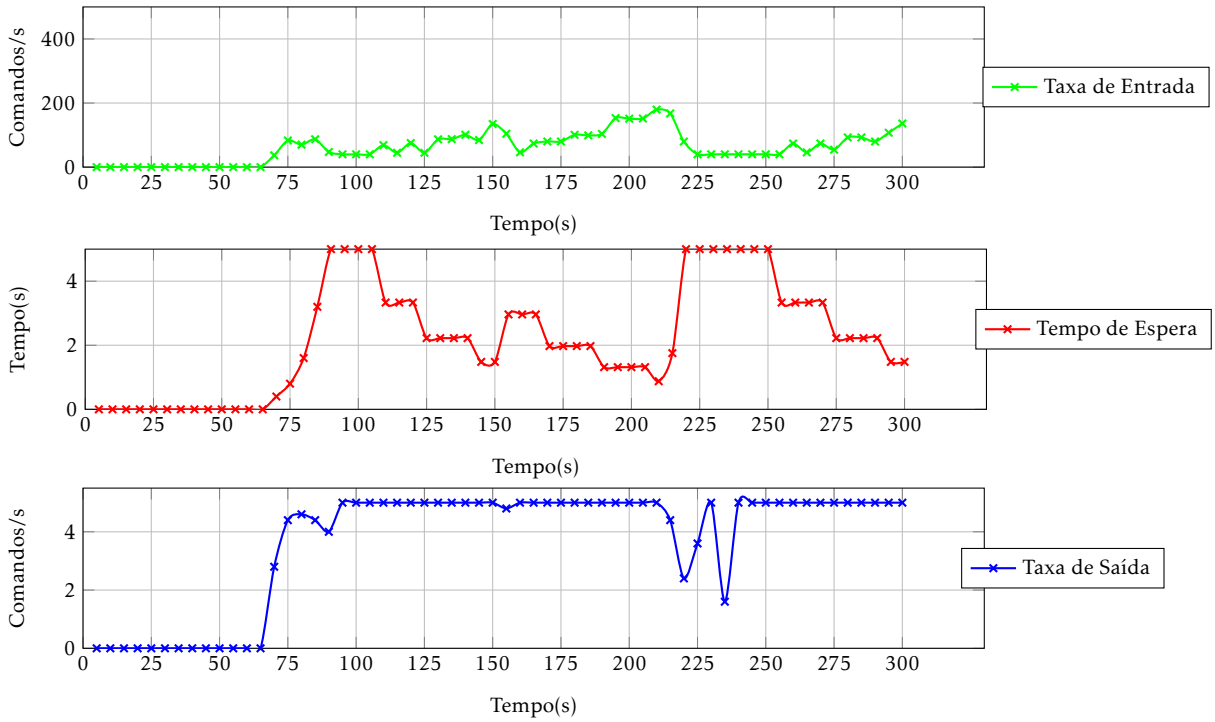

Figura 4.19: Algoritmo tempo médio de processamento de 50ms, 1 servidor, 200 clientes

forma igual pelos dois servidores.

O servidor de nível 1 tem como função fornecer comandos à aplicação com frequência pretendida de 5 comandos por segundo, o que significa que sempre que a taxa de saída seja inferior a 5, o tempo de espera deve de aumentar.

O segundo servidor, de nível 2, fornece comandos ao servidor acima, inicialmente com a mesma frequência. Este valor é depois adaptado às necessidades do servidor de nível 1.

Na Figura 4.20 está representado o caso mais simples, em que estão conectados 50 controladores ao sistema, divididos igualmente pelos dois servidores. O número de comandos recebidos em média, por segundo, por todo o sistema, ou seja, pelos 2 servidores em conjunto é superior a 300. O tempo de espera foi sempre mínimo, nunca superando os 0.5 segundos.

Em seguida, aumentou-se o número de controladores conectados, passando a ser 100 controladores, e o resultado está exposto na Figura 4.21. Apesar do aumento controladores conectados, o tempo de espera manteve níveis muito baixos. Nenhum dos servidores superou os 0.6 segundos. As taxas de entrada médias somadas, rondam os 550 comandos por segundo

Enquanto nos casos menos exigentes, o tempo de espera manteve-se em níveis muito baixos, nos casos com 150 e 200 controladores, representados pelas Figuras 4.22 e 4.23 respetivamente, houve oscilações grandes de forma a que a taxa de saída por parte do servidor de nível 1 fosse alta o suficiente para satisfazer as necessidades da aplicação destino.

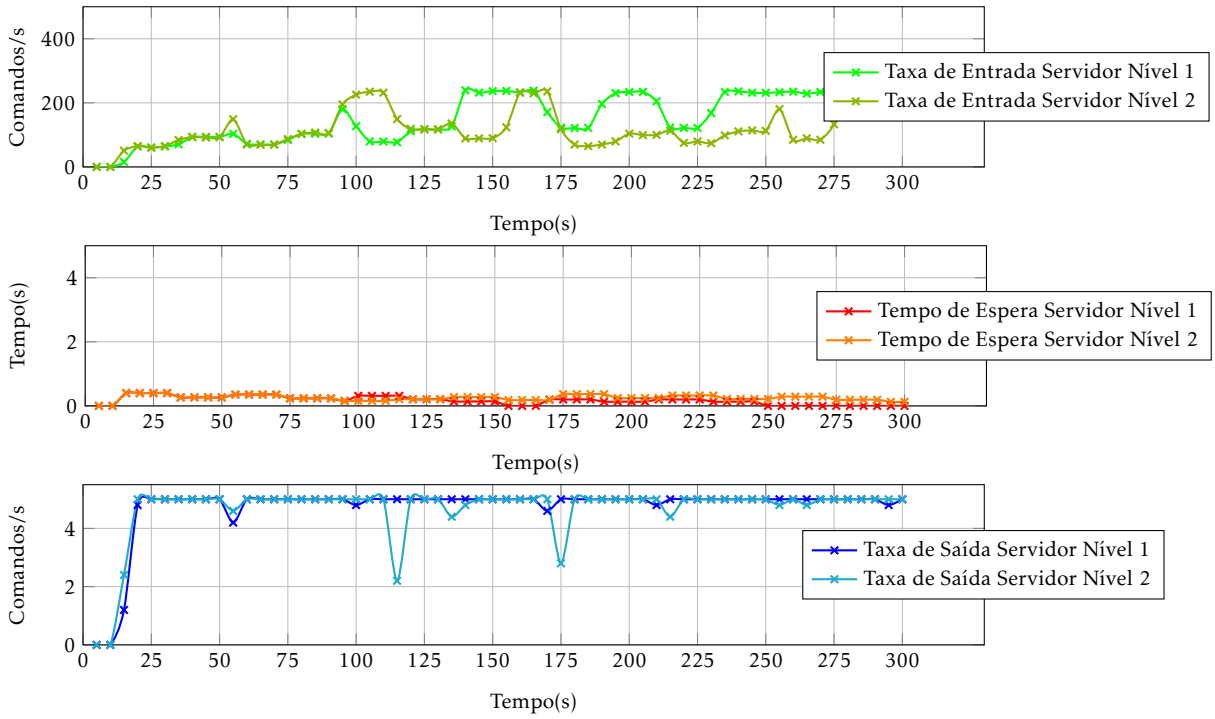

Figura 4.20: Algoritmo tempo médio de processamento de 25ms, 2 servidor, 50 clientes (25/25)

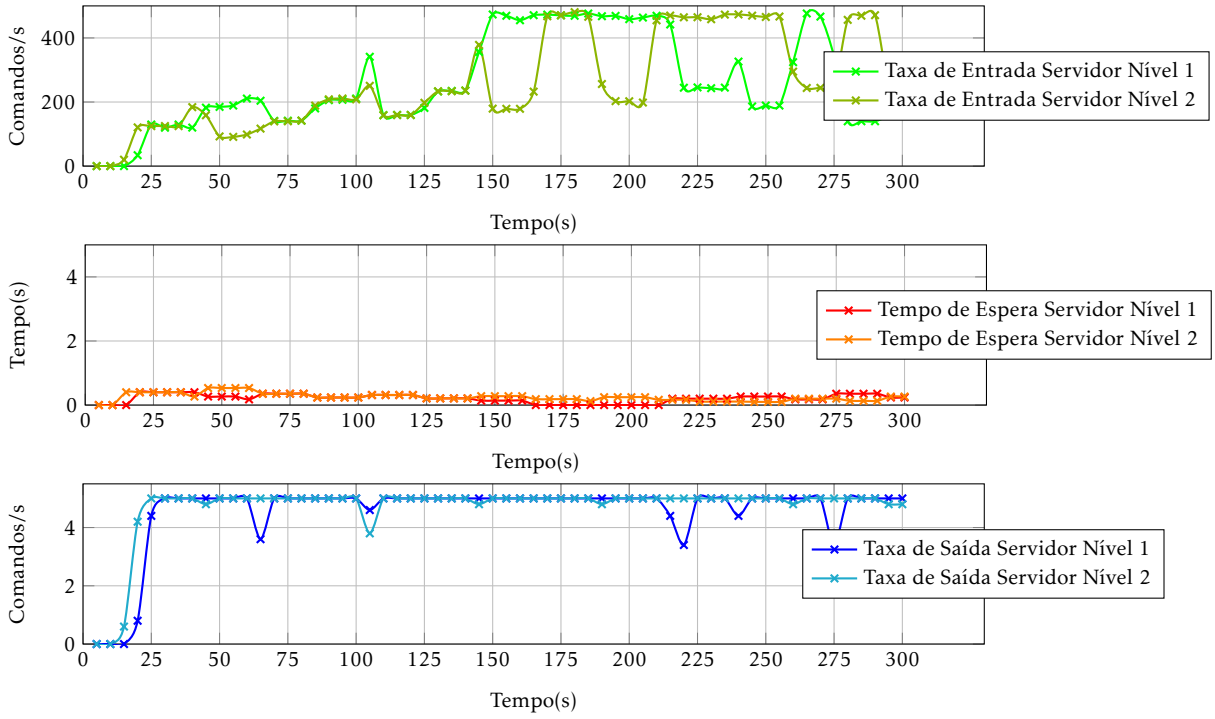

Figura 4.21: Algoritmo tempo médio de processamento de 25ms, 2 servidores, 100 clientes (50/50)

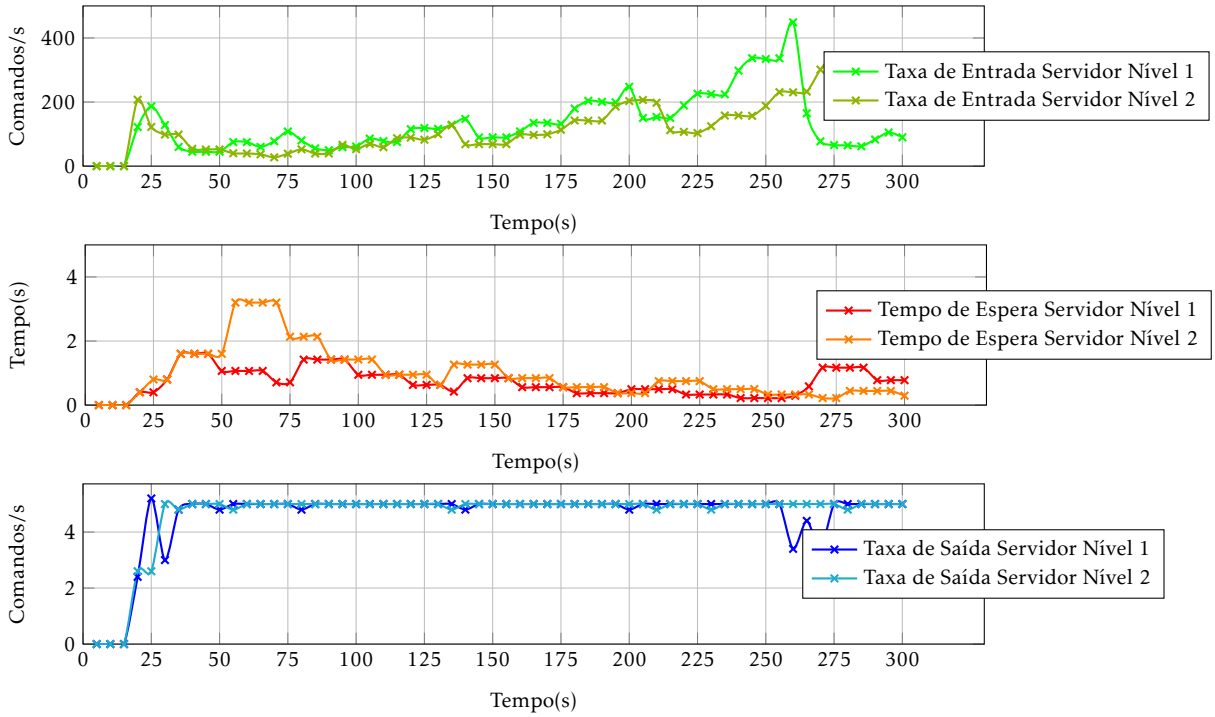

Figura 4.22: Algoritmo tempo médio de processamento de 25ms, 2 servidores, 150 clientes (75/75)

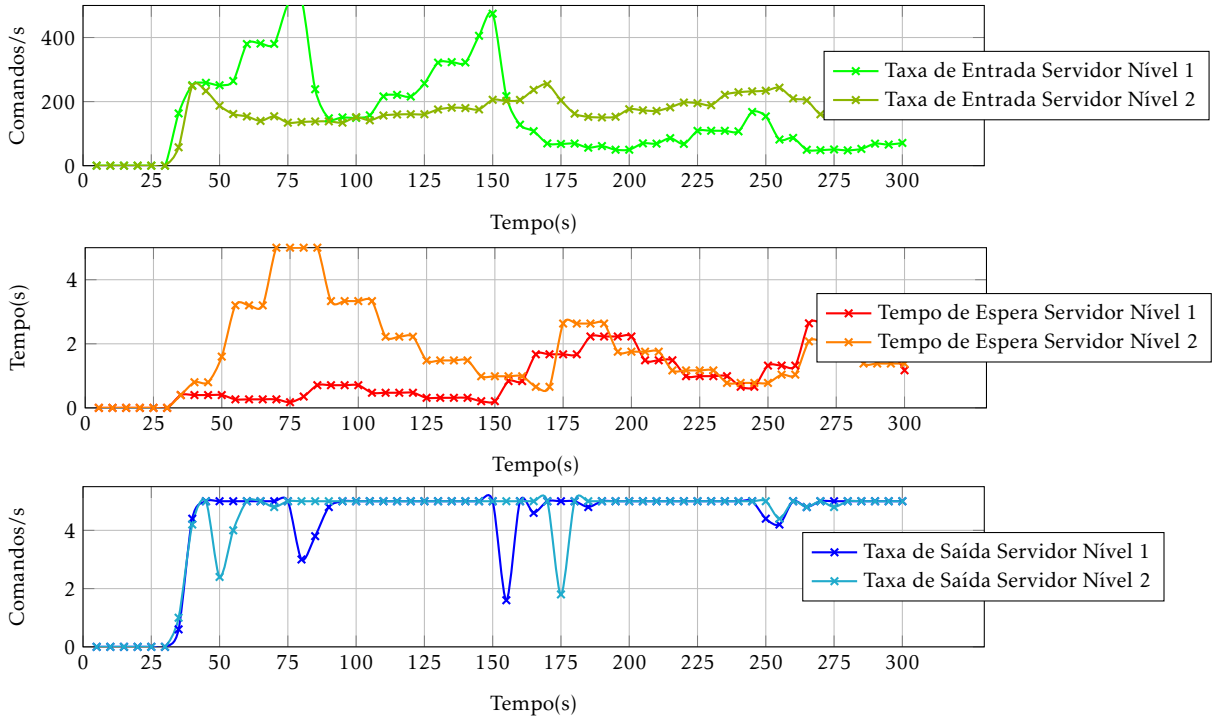

Figura 4.23: Algoritmo tempo médio de processamento de 25ms, 2 servidores, 200 clientes (100/100)

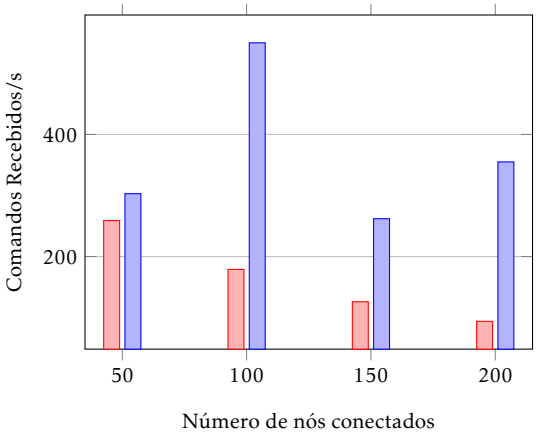

Figura 4.24: Comparação entre a utilização de um ou dois servidores

Na Figura 4.24, é feita a comparação entre a utilização de um ou dois servidores. O gráfico, representa o número médio de comandos recebidos, ou seja, no caso em que existem dois servidores, representados pela cor azul, o número de comandos recebidos apresentados na figura é a soma da média dos comandos recebidos pelos dois servidores.

Analisando o gráfico, podemos concluir que houve uma melhoria significativa quando são utilizados dois servidores. No caso com menos controladores conectados, as melhorias não são muito visíveis, pois o número de comandos recebidos é semelhante. Contudo, nos outros casos, se observarmos o gráfico da taxa de entrada (à direita), percebemos que houve um aumento de pelo menos 50% de comandos recebidos.

Quanto à taxa de saída, o sistema conseguiu manter sempre valores aproximados da frequência desejada. À medida que o número de dispositivos conectados cresce, aumenta o tempo de espera dos controladores, fazendo com que o número de comandos recebidos desça. Esta afirmação é bem visível no gráfico se compararmos o caso de 100 e 150 controladores.

Por fim, fez-se um teste em que foram utilizados 3 servidores, em que dois deles recebiam comandos vindos dos controladores e o terceiro, o servidor principal (nível 1), apenas combina os resultados vindos dos servidores abaixo e os fornece à aplicação destino. Os resultados referente à taxa média de entrada (à esquerda) e saída de comandos (à direita) estão representados nas Figura 4.25.

Como o servidor principal apenas recebe as informações dos outros servidores, ou seja, não recebe quaisquer comandos, apenas fluxos processados, não foi contabilizada a sua taxa de entrada. Os dois servidores secundários estão representados a vermelho e a azul, enquanto o servidor de nível 1 está representado a laranja.

Os resultados foram idênticos aos obtidos apenas com dois servidores, contudo o servidor principal, representado a laranja, na Figura 4.25, envia sempre, o mesmo número de comandos para a aplicação. Isto deve-se ao facto de não precisar de processar qualquer informação. Mas, ao observarmos o comportamento do sistema apenas com 2 servidores, obtém-se resultados muito semelhantes, isto é, o número de comandos recebidos é o

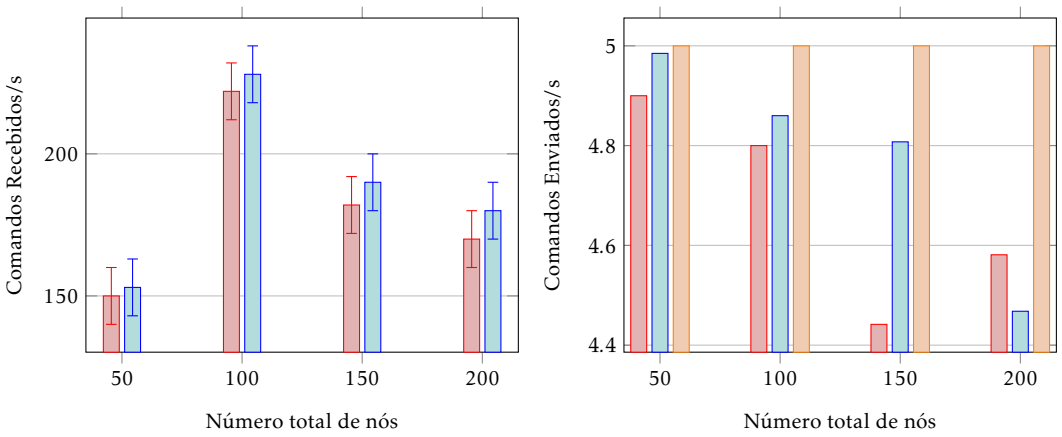

Figura 4.25: Taxa de Entrada e Saída com 3 Servidores

mesmo, a taxa de saída é bastante próxima da pretendida.

Sendo assim, esta abordagem de os servidores estarem organizados em pirâmide, como na Figura 2.4 não é desejada, desta forma poupam-se recursos, pois existe menos um servidor e existe menos um ponto de falha na rede. A abordagem recomendada é então a que está ilustrada na Figura 3.1.

4.5.3 Discussão

A aplicação desenvolvida comporta-se da maneira esperada, enviando comandos sempre que o controlador executar movimentos com o seu dispositivo. O consumo energético de um controlador é baixo o que permite a sua utilização em eventos de curta e média duração, como o caso de um jogo.

Ao testar o comportamento de um controlador como servidor, conseguimos perceber que é uma opção viável devido ao seu tempo de processamento médio. Suporta um número considerável de controladores (foram feitos testes com 25 controladores conectados) e a sua taxa de saída é muito perto da desejada pela aplicação. O seu consumo energético como servidor é um pouco maior de um controlador comum, mas mesmo assim, são números aceitáveis para a maioria dos dispositivos presentes no mercado, podendo facilmente ser utilizada esta forma de processamento em eventos de média duração.

Ao observarmos o comportamento do sistema quando processa o nosso caso de estudo, observamos que este tem o comportamento desejado, fornecendo ao nosso jogo comandos suficientes de forma a que este decorra dentro da normalidade. Mesmo com um grande número de dispositivos conectados (200), a taxa de saída mantém-se sempre elevada. É importante realçar que neste caso o sistema chega a processar mas de 1000 comandos por segundo.

Aumentando a carga de processamento, o sistema tem maior necessidade de se adaptar e percebe-se ao observar casos como o da Figura 4.19 que apesar de haver um grande número de dispositivos ligados o número de comandos que o sistema recebe é muito baixa devido ao tempo de espera que o sistema impõe aos controladores. Isto acontece

porque o sistema dá prioridade à taxa de saída, sendo que os valores obtidos são sempre muito aproximados do pretendido.

Para melhorar o comportamento do sistema, ou seja, para que sejam processados mais comandos, tem de se aumentar o número de servidores e podemos concluir que o sistema é escalável, pois com um maior número de servidores o número de comandos recebidos aumentou consideravelmente, como podemos observar na Figura 4.24.

Capítulo 5

Conclusões

Este capítulo conta com duas secções, a primeira apresenta as conclusões sobre esta dissertação e em seguida uma secção com possíveis melhorias do sistema desenvolvido.

5.1 Conclusões

Nesta dissertação foi desenvolvida uma *framework* genérica que permite a cooperação entre vários dispositivos móveis. O sistema desenvolvido permite uma nova forma de interação entre utilizadores, fornecendo assim uma experiência diferente em eventos sociais.

Para além do *middleware*, foi também desenvolvido um controlador de forma a providenciar comandos ao sistema.

Sendo assim, o sistema permite aglomerar fluxos de comandos emitidos por vários controladores, resultando num comando final e único a utilizar numa aplicação independente de todo o processo.

A abordagem para a aglomeração de fluxos consiste em separar-los em pequenos segmentos e dividir-los por várias *threads* de processamento. O resultado vindo de cada *thread* é também o comando a ser enviado para a aplicação destino.

Para validar este conceito, foi utilizado para caso de estudo, uma aplicação, semelhante ao jogo do *Pong*, desenvolvido pela Faculdade de Ciências e Tecnologias e através da utilização dessa aplicação foi possível verificar o bom funcionamento do sistema.

Com esta aplicação foi ainda possível recolher resultados acerca do consumo energético e ficou comprovado que este sistema pode ser utilizado em eventos de curta/média duração, devido ao seu baixo consumo na execução das suas tarefas.

Para além dessa aplicação, foi ainda testado em ambiente simulado, com um algoritmo

mais exigente daquele que foi utilizado no caso de estudo referido acima, em que foi possível verificar as possíveis falhas que o sistema pode ter. Essas falhas são principalmente a baixa taxa de entrada de comandos o que faz com que os controladores tenham pouco impacto na aplicação. Essa situação pode ser melhorada com o aumento dos servidores disponíveis. Outra das dificuldades encontradas é a forma de como os servidores se adaptam, apesar das taxas de saída serem muito aproximadas dos valores desejados, a forma de adaptação dos servidores pode ser ainda melhorada.

Ficou também provado que a utilização de uma estrutura em pirâmide não é a melhor solução, ou seja, todos os servidores que compõe a rede, devem receber comandos vindos de controladores de modo a evitar-se desperdício de hardware.

Concluindo, os objetivos desta dissertação foram cumpridos, com a materialização de um protótipo do *middleware* que permite a cooperação de dispositivos móveis. O sistema foi avaliado principalmente em ambiente simulado ficando apenas a faltar uma avaliação real com um número grande de utilizadores conectados. Contudo, através dessa avaliação virtual, foi possível comprovar que este sistema é uma possível solução para eventos sociais em que se pretende uma forma de cooperação, cumprindo assim as expetativas previstas para esta dissertação.

5.2 Trabalho Futuro

O sistema desenvolvido é um protótipo funcional, podendo ser utilizado com qualquer aplicação destino. No entanto, existem alguns problemas que podem ser melhorados e ainda algumas questões que ficaram por resolver. O trabalho que ainda pode ser desenvolvido de forma a melhorar o sistema consiste em:

Utilização de uma janela deslizante: Os fluxos de comandos são apenas discretizados por um intervalo de tempo fixo, contudo, uma das possíveis abordagens, tal como foi discutido na Secção 2.3.3, é a utilização de uma janela deslizante de modo a suavizar os resultados, de forma a que os comandos finais sejam mais homogéneos.

Heurística de Adaptação: A forma de como os servidores se adaptam quando a carga é muito grande é simples: quando faltam comandos é pedido aos controladores conectados que reduzam a frequência com que enviam comandos e caso contrário, que aumentem a frequência. Contudo, esta abordagem pode ser melhorada, desenvolvendo-se uma heurística que regule melhor os valores da frequência de receção de comandos.

Conexão aos servidores No protótipo desenvolvido, a conexão aos servidores foi um tema debatido mas não lhe foi dado grande importância, tendo isto em conta, os controladores apenas se conectam a um servidor e tem como prioridade o servidor menos cheio. No caso real, outros fatores têm de ser contabilizados, como localização geográfica, latência, ou até, se este servidor for também um controlador, bateria

do dispositivo. Sendo assim, uma forma de melhorar o protótipo concebido será conectar ao servidor tendo em conta os fatores enumerados anteriormente.

Tolerância a falhas Neste momento, o sistema desenvolvido não conta com possíveis falhas no sistema. Futuramente poderia ser implementada uma forma de replicar a informação pelos vários servidores de forma a ser possível recuperar de falhas.

Validação do protótipo: De forma a validar o protótipo desenvolvido, seria interessante em ver este sistema utilizado num ambiente real com um número de pessoas na ordem das dezenas ou centenas, de modo a retirar-se outro tipo de conclusões. Para além de que, este protótipo apenas foi testado recorrendo ao jogo do *Pong* descrito no caso de estudo, porém pode ser utilizado noutros jogos ou eventos e seria interessante verificar se o comportamento seria o mesmo.

São estas as principais lacunas do sistema, que podem ser melhoradas em trabalho futuro.

Bibliografia

[1] Cisco. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021 White Paper*. 2017. URL: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html.

[2] Wi-Fi Alliance. *Wi-Fi*. Accessed: 2017-12-26. URL: http://www.wi-fi.org/.

[3] Bluetooth SIG. *"Topology Options"*. Accessed: 2017-12-26. URL: https://www.bluetooth.com/bluetooth-technology/topology-options.

[4] OnePlus. *OnePlus 6*. Accessed: 2018-08-26. URL: https://www.oneplus.com/pt/6.

[5] J. A. Silva, H. Paulino, J. M. Lourenço, J. Leitão e N. M. Preguiça. "Time-Aware Publish/Subscribe for Networks of Mobile Devices". Em: *CoRR* abs/1801.00297 (2018).

[6] F. Cerqueira, J. A. Silva, J. M. Lourenço e H. Paulino. "Towards a persistent publish/subscribe system for networks of mobile devices". Em: *MECC@Middleware*. ACM, 2017, 2:1–2:6.

[7] T. Vajk, P. Coulton, W. Bamford e R. Edwards. "Using a mobile phone as a "wii-like" controller for playing games on a large public display". Em: *International Journal of Computer Games Technology* 2008 (2008).

[8] J. Leikas, H. Stromberg, V. Ikonen, R. Suomela e J. Heinila. "Multi-user mobile applications and a public display: novel ways for social interaction". Em: *Fourth Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM'06)*. 2006, 5 pp.–70. DOI: 10.1109/PERCOM.2006.38.

[9] The Centre for Computing History. *"Atari PONG"*. Accessed: 2018-01-16. URL: http://www.computinghistory.org.uk/det/4007/Atari-PONG/.

[10] Atari. *Atari*. Accessed: 2018-08-26. URL: https://www.atari.com/.

[11] ArcadeClassics.net. *"Arkanoid: Classic Arcade Game Video, History and Game Play Overview"*. Accessed: 2018-01-16. URL: https://arcadeclassics.net/80s-game-videos/arkanoid/.

[12] P. Narasimhan e F. Silva. *Hyrax: Crowd-Sourcing mobile devices to develop edge cloud*. Accessed: 2018-01-25. 2017. URL: http://hyrax.dcc.fc.up.pt.

[13] Oracle. *"Java"*. Accessed: 2018-08-26. URL: https://www.java.com.

[14] Google. *"Android"*. Accessed: 2018-08-26. URL: https://www.android.com/.

[15] Sony. *"Comando sem fios DUALSHOCK 4"*. Accessed: 2018-08-26. URL: https://www.playstation.com/pt-pt/explore/accessories/dualshock-4-wireless-controller/.

[16] Nintendo. *"Wii U Accessories"*. Accessed: 2018-08-26. URL: https://www.nintendo.com/wiiu/accessories.

[17] G. W. Young, A. Kehoe e D. Murphy. "Usability testing of video game controllers". Em: *Games User Research: A Case Study Approach (2016)* 145 (2016).

[18] N. Katzakis, M. Hori, K. Kiyokawa e H. Takemura. "Smartphone game controller". Em: *Proceedings of the 74th HIS SigVR Workshop*. Citeseer. 2011.

[19] Android Developers. *"Sensor Overview"*. Accessed: 2017-12-26. URL: https://developer.android.com/guide/topics/sensors/sensors_overview.html.

[20] Android Developers. *"Motion Sensors"*. Accessed: 2017-12-26. URL: https://developer.android.com/guide/topics/sensors/sensors_motion.html.

[21] Android Developers. *"Button"*. Accessed: 2017-12-26. URL: https://developer.android.com/reference/android/widget/Button.html.

[22] Bluegiga Technologies. *"Classic Bluetooth vs Bluetooth low energy"*. Accessed: 2018-01-10. URL: http://www.mt-system.ru/sites/default/files/docs/documents/bluetooth_le_comparison.pdf.

[23] G. V. Zaruba, S. Basagni e I. Chlamtac. "Bluetrees-scatternet formation to enable Bluetooth-based ad hoc networks". Em: *communications, 2001. ICC 2001. IEEE International Conference on*. Vol. 1. IEEE. 2001, pp. 273–277.

[24] Android Developers. *"Bluetooth"*. Accessed: 2017-12-26. URL: https://developer.android.com/guide/topics/connectivity/bluetooth.html.

[25] Oracle Corporation and/or its affiliates. *"Java Community Process"*. Accessed: 2017-12-26. URL: https://www.jcp.org.

[26] BlueCove Team. *"BlueCove"*. Accessed: 2017-12-26. URL: http://bluecove.org/.

[27] Android Developers. *"android.net.wifi"*. Accessed: 2017-12-26. URL: https://developer.android.com/reference/android/net/wifi/package-summary.html.

[28] Oracle Corporation and/or its affiliates. *"NetworkInterface"*. Accessed: 2017-12-26. URL: https://docs.oracle.com/javame/8.0/api/gcf/api/javax/microedition/io/NetworkInterface.html.

[29] Wi-Fi Alliance. *Wi-Fi Direct*. Accessed: 2017-12-26. URL: http://www.wi-fi.org/discover-wi-fi/wi-fi-direct/.

[30] A. Teófilo, D. Remédios, H. Paulino e J. Lourenço. "Group-to-Group Bidirectional Wi-Fi Direct Communication with Two Relay Nodes". Em: *MobiQuitous*. ICST / ACM, 2015, pp. 275–276.

[31] J. A. Silva, J. Leitão, N. M. Preguiça, J. M. Lourenço e H. Paulino. "Towards the Opportunistic Combination of Mobile Ad-hoc Networks with Infrastructure Access". Em: *MECC@Middleware*. ACM, 2016, p. 3.

[32] Valve. *Source Multiplayer Networking*. Accessed: 2017-12-26. URL: http://orca.st.usm.edu/~seyfarth/network_pgm/net-6-3-3.html/.

[33] J. Rodrigues, E. R. Marques, L. Lopes e F. Silva. "Towards a middleware for mobile edge-cloud applications". Em: *Proceedings of the 2nd Workshop on Middleware for Edge Clouds & Cloudlets*. ACM. 2017, p. 1.

[34] K. Antonis, J Garofalakis, I. Mourtos e P Spirakis. "A hierarchical adaptive distributed algorithm for load balancing". Em: *Journal of Parallel and Distributed Computing* 64.1 (2004), pp. 151–162.

[35] S. Gargolinski, C. St Pierre e M. Claypool. "Game server selection for multiple players". Em: *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*. ACM. 2005, pp. 1–6.

[36] Y. Yu, P. K. Gunda e M. Isard. "Distributed aggregation for data-parallel computing: interfaces and implementations". Em: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 247–260.

[37] A. S. Foundation. *MapReduce Tutorial*. Accessed: 2018-01-25. URL: https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[38] W. Ren, R. W. Beard e E. M. Atkins. "Information consensus in multivehicle cooperative control". Em: *IEEE Control Systems* 27.2 (2007), pp. 71–82.

[39] J. Gama e C. Pinto. "Discretization from data streams: applications to histograms and data mining". Em: *Proceedings of the 2006 ACM symposium on Applied computing*. ACM. 2006, pp. 662–667.

[40] M. Stonebraker, U. Çetintemel e S. Zdonik. "The 8 Requirements of Real-time Stream Processing". Em: *SIGMOD Rec.* 34.4 (dez. de 2005), pp. 42–47. ISSN: 0163-5808. DOI: 10.1145/1107499.1107504. URL: http://doi.acm.org/10.1145/1107499.1107504.

[41] Microsoft. *"Azure Stream Analytics"*. Accessed: 2018-08-26. URL: https://azure.microsoft.com/pt-pt/services/stream-analytics/.

[42] Google Developers. *"Protocol Buffers"*. Accessed: 2018-08-26. URL: https://developers.google.com/protocol-buffers/.

[43] A. Sumaray e S. K. Makki. “A comparison of data serialization formats for optimal efficiency on a mobile platform”. Em: *Proceedings of the 6th international conference on ubiquitous information management and communication*. ACM. 2012, p. 48.

[44] Android Developers. *"TYPE ROTATION VECTOR"*. Accessed: 2018-08-26. URL: https://developer.android.com/reference/android/hardware/Sensor.html.

[45] Java. *"Class LinkedList<E>"*. Accessed: 2018-08-26. URL: https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html.

[46] C. Bunse, H. Höpfner, S. Roychoudhury e E. Mansour. “Choosing the"Best"Sorting Algorithm for Optimal Energy Consumption.” Em: *ICSOFT (2)*. 2009, pp. 199–206.

APÊNDICE A

Exemplo de especialização da framework

Listagem A.1: Especialização da Framework

```
1  public class Builder {
2
3      public static void main(String[] args) {
4
5          MyApp myApp = new MyApp(){
6              @Override
7              public void send(R result) {
8                  // make connection to app and send the result
9              }
10         };
11
12         Algorithm alg1 = new Algorithm() {
13             @Override
14             public Pair process(CommandList commandList,
15              CommandList processedCommands, int weight) {
16                 //choose algoritm1
17                 return null;
18             }
19         };
20
21         Algorithm alg2 = new Algorithm() {
22             @Override
23             public Pair process(CommandList commandList,
24              CommandList processedCommands, int weight) {
25                 //choose algoritm2
26                 return null;
27             }
28         };
29
```

```
30          Map<String, Algorithm> algorithmMap = new HashMap<>(2);
31          algorithmMap.put("type1", alg1);
32          algorithmMap.put("type2", alg2);
33
34          Connection myConnection = new Connection(){
35
36              @Override
37              public void enableVisible() {
38
39              }
40
41              @Override
42              public void discover() throws IOException {
43
44              }
45
46              @Override
47              public void connect(String address) {
48
49              }
50
51              @Override
52              public void send(float value, int weight,
53              String type, String team) {
54
55              }
56
57              @Override
58              public void shutDown() {
59
60              }
61
62              @Override
63              public SettingsClass.Settings getSettings() {
64                  return null;
65              }
66
67              @Override
68              public long getWaitTime() {
69                  return 0;
70              }
71
72              @Override
73              public int getMyLvl() {
74                  return 0;
75              }
76          };
77
78          Settings mySettings = new Settings();
79          mySettings.setFrequency(200);
```

```
80          mySettings.setAppName("App Name");
81
82          CooperativeServer myServer = new CooperativeServer(myApp, algorithmMap,
83          myConnection, mySettings);
84      myServer.start();
85      }
86  }
```

Diogo Pinhal Ribeiro

Licenciado em Ciência e Engenharia Informática

Middleware para controlo cooperativo a partir de uma rede de dispositivos móveis

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Setembro, 2018

Diogo Pinhal Ribeiro

Licenciado em Ciência e Engenharia Informática

Middleware para controlo cooperativo a partir de uma rede de dispositivos móveis

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Setembro, 2018

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Diogo Ribeiro 2018
Middleware para controlo cooperativo a partir de uma rede de dispositivos móveis 2018